(1)

AFIT/GCS/ENG/93D-11

ALTERNATIVE ARCHITECTURES FOR DOMAIN-ORIENTED APPLICATION COMPOSITION AND GENERATION SYSTEMS

THESIS

Warren Evan Gool
Captain, USAF

AFIT/GCS/ENG/93D-11

AFIT/GCS/ENG/93D-11

Alternative Architectures for Domain-Oriented Application Composition and Generation Systems

THESIS

Presented to the Faculty of the Graduate School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science

Warren Evan Gool, B.S.C.S.

Captain, USAF

December, 1993

Acknowledgements

I would like to thank my thesis committee for their support and advice on this arduous undertaking. A special thanks goes out to my advisor, Maj Bailor, for constantly "pulling me out of the trees so that I could see the forest"; he continually made time on short notice to give me both council and guidance.

For being able to leave AFIT with my sanity intact and a sense of humor (warped though it may be), my hat goes off to the morning coffee crowd. Jay Cossentine, Jeff Miller, and John Keller were a sounding board for my continuous whining and barrage of bad jokes.

I am especially appreciative of the patience of my brother and sister, Bart and Norma, for understanding that I could not be home with them when my family needed me the most; I truly believe they realize that it was better for me to be here during the summer of 1993.

And next I thank my endearing wife, who singlehandedly managed [illegible]e household and chores while I was camping at AFIT. She was forever surprising me wi[illegible] her patience and understanding of my lack of time to spend with her. Her unswerving love through this trial has only strengthen my love for her. I am forever indebted to you, Teresa

Finally, I dedicate this effort in loving memory of my mother, Esther Whalen Gool. I believe she is the one who gave me the strength to persevere this experience. I can remember her often saying "An education is the key to making your dreams come true."

Warren Evan Gool

Table of Contents

List of Figures

Figure Page

List of Tables

AFIT/GCS/ENG/93D-11

Abstract

This thesis presents a formalized framework for comparing the structure and semantics of software architectures. The framework uses object diagrams for analyzing the structure of the architectures and the axiomatic approach for analyzing the semantics. This framework is used to compare the Object Connection Update (OCU) model (developed by the Software Engineering Institute) against four other software architectures: VHDL defined by Lipsett, MetaH defined by Honeywell, μRapide defined by Luckham, and hierarchical software systems as defined by Batory. The goal of the comparison was to evaluate the OCU model for suitability within prototype application composition and generation systems. This research concluded that the OCU model has all the elements necessary for use in application composition and generation systems. Additionally, the framework identified several common elements in all the software architectures. These common elements may lead to the development of a "meta-model" for software architectures.

Alternative Architectures for Domain-Oriented Application Composition and Generation Systems

I. Introduction and Problem Statement

1.1 Background

Software engineering will not become an engineering discipline until software engineers develop software systems the way engineers develop systems. Key elements of an engineering discipline are found in how engineers formulate problems, use technology bases, perform design by composing components and produce solutions (10). The most prominent engineering element missing from software engineering is the extensive amount of reuse from a technology base. Public documentation in a technology base is reviewed for possible reusable components and the necessary components are extracted to achieve a design. Engineers also have a way of measuring the success of the design before construction actually begins. This success is measured by employing a scientific, theoretical foundation to verify by systematic calculation that a proposed design satisfies the specifications. Once a design is constructed and confidence in it grows, it is added to the public knowledge base for future reuse (14). This is how engineering has sustained its level of development over the years.

In contrast, software engineers currently do not employ techniques relying on the composition of reusable components, a published knowledge base, or methods for measuring success prior to development (3). By starting from scratch each time, a software engineer spends extensive amounts of time debugging the whole system. If the analyst had employed reusability (via composition), testing would be reduced, since the only area that would need to be tested is the overall design (given that the individual components came from a certified technology base). This would preclude testing down to the individual component, since the component came from a knowledge base of established confidence. Furthermore, if the analyst used formal methods as a means of verifying the overall system design (by use of proofs), confidence in the system could be established prior to system development.

As mentioned above, engineers reuse components. These components are nothing more than models. Models can be thought of as a codified body of scientific knowledge and technology presented in a (re)usable form (10). As such, models have a way to influence or interface with their environment. Thus, models are merely representations of real world objects. These models are what an engineer extracts from a public knowledge base to aid in design. At present, software engineers are just discovering the advantages of reuse by employing object-oriented analysis and design. It has been shown that software developed using the object-oriented methodology has very high cohesion within the object (12). Further, these systems demonstrate very loose coupling between the objects (12). The loose coupling of the objects and high cohesion within an object have made it easier to reuse software components.

Part of designing a software system requires an analysis of the domain of the problem space by the software engineer. Prior to systems analysis, the software engineer creates a model of the software system, trying to generalize all subsystems in the application domain by means of a domain model that transcends specific applications. If this is successful, the next step is to define a domain-specific language. This language becomes the model and is used to describe objects and operations common to that domain. A more formal definition would be "a language with syntax and semantics designed to represent all valid actions and objects in a particular domain" (19).

During system design, the software engineer chooses a software architecture that logically follows from the domain model. Webster's dictionary defines architecture as "a method or style of building." To apply this definition to a software system would imply "the way objects (models) are composed is defined by an architecture." Another way to define software architecture is that a software architecture imposes a uniform style on the structure of the software system (1:110). A software architecture is a selection (from a technology base/public knowledge base) of models and composition rules that defines the structure, performance, and use of a system relative to a set of engineering goals (14:8). Thus, an architecture is a way of composing models, using engineering practices to achieve an overall system design.

While the software architecture serves as the framework for the design, this concept is insufficient by itself for supplying the additional details required for a specific design. Additional domain knowledge is still needed to instantiate components of the architecture and develop optimized algorithms for the problem domain. These details may be provided by use of the domain-specific language. Thus, the general concept of a software architecture and the specific design details provided by domain-specific languages are combined to create what can be termed a domain-specific software architecture (DSSA) (5).

So, until software engineers employ extensive reuse of software components (such as models or objects), incorporate a means of verifying a design prior to implementation (formal methods), and establish a software architecture that easily incorporates reusable components, they will continue to produce software systems with large amounts of errors. These systems will continue to take a large amount of resources during development and may or may not meet user requirements.

1.2 Problem

To help push software engineering into more of a true engineering discipline, the Air Force Institute of Technology (AFIT) has prototyped a domain-oriented application composition and generation system called Architect. It is based on integrating the concepts of software architectures and formal domain models. The purpose of Architect is to allow an application specialist to input system specifications in a domain-specific language. Application specialists are sophisticated "users"; they are familiar with the overall domain and understand what the new application must do to meet requirements; they provide the application-specific information needed to specify an application (20:3-2). After verifying that the new composition behavior meets the specifications, application specialists can have Architect synthesize the appropriate implementation of the specification. Synthesizing a specification is the transforming of the specification into the language of the target system.

At the center of Architect is a technology base used to store domain and software engineering information. This technology base consists of formal specifications developed using the Refine formal specification language. The software architecture of Architect has been implemented using the Object Connection Update model proposed by the Software Engineering Institute (SEI) (14). Architect's initial implementation did not incorporate all of the OCU methodology. One of the focuses of this thesis effort is on adding additional functionality to Architect. Architect's new functionality and corresponding implementation details are presented in subsequent chapters.

Another focus of this research is on the suitability of the OCU model for the software architecture of Architect. The software architectural model within Architect must be flexible enough to allow for future enhancements, as well as provide the capability to accommodate a technology base with components from different domains. To aid in the establishment of the technology base, Architect should allow for the inclusion of artifacts from other domains. However, prior to allowing artifacts from other domains, a mapping between software architectures used in different domains must be established. The first step in developing a mapping between software architectures is to establish a framework for comparing software architectures.

1.2.1 Problem Statement.

Develop a formalized model which serves as a framework for comparing software architectures. Include in this framework a consistent standard for analyzing software architectural structures and analyzing the semantics of the architectural components.

1.3 Scope

The scope of this research includes the development of a standard framework for comparing software architectures based on:

1. Object Diagrams, as presented by Rumbaugh (22), as the representation for comparing structural components of an architecture. I will not specify complete object models with methods and attributes per (22), as the focus of this thesis is on architectural entities and their relationships.

2. An axiomatic approach comparing the semantics of the components that appear similar between architectures (4, 11). Additionally, the semantic analysis of the architectural components is limited to identifying the preconditions and postconditions that must be maintained as some abstract program is executed. The development of a standardized abstract program language and corresponding execution function is left for future research.

This thesis also includes the implementation of an event-driven simulation capability within the Architect prototype. More specifically, I address the changes that need to be made to the OCU architectural model and its simulation capabilities within Architect to support an event-driven mode of execution. These changes are limited to identifying structural changes to the subsystems and primitives, as well as the execution capabilities of the subsystems and the primitives within Architect. It does not include the development of an application executive, as that is addressed in a separate effort (29). In essence, the application executive provides an operating system type environment with services.

1.4 Big Picture

Figure 1.1 shows the overall research effort that is being conducted at AFIT. My particular area of research is isolated to the Application Composer block in the lower left hand corner.

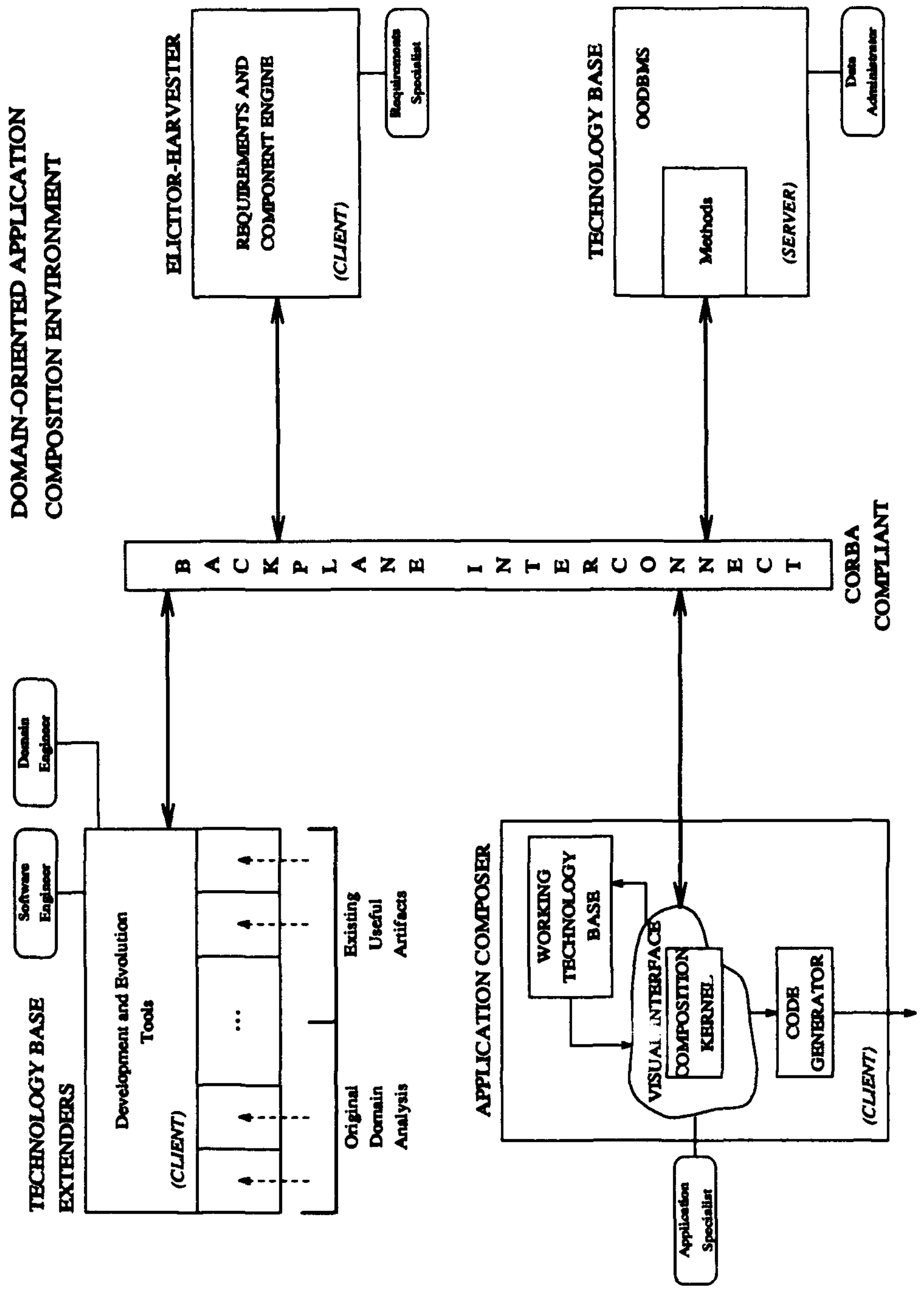


Figure 1.1 Architect's Software Composition and Generation Toolset

1.5 Sequence of Presentation

The order of presentation for the rest of this thesis is as follows. Chapter II provides some example definitions for a software architecture. It also presents some identifiable software architectures used in industry today. Chapter III details requirements for comparison of software architectures, as well as the concept of operations for an event-driven simulation capability within Architect. Chapters IV and V present the methodology for comparing the alternative software architectures and the results of employing this methodology on the five architectures. Chapter VI provides detailed design of the event-driven capability within Architect, while Chapter VII shows how the event-driven capability was validated. Finally, the results and conclusions for this research effort are presented in Chapter VIII.

II. Literature Search

2.1 Introduction

The previous chapter presented Webster's definition of architecture as: "a method or style of building". Also presented were several definitions for software architecture. In this thesis, a software architecture will be defined as "...a selection (from a technology base/public knowledge base) of models and composition rules that defines the structure, performance, and use of a system relative to a set of engineering goals" (4). In this chapter, I examine some identifiable software architectures. Next, I discuss Domain-Specific Software Architecture and define all the terms associated with this architectural model. Finally, I provide examples of the use of some of the different software architectures.

2.2 Software Architectures

The following sections present a description of some of the common software architectures used in system design.

2.2.1 Layered Architecture. Software designed as an hierarchical collection of components is a layered architecture; each level of the software is built using the software components from a lower layer. Ideally, components at each layer are independent of each other. Figure 2.1 presents a conceptual model of a layered architecture. The communication in a layered architecture is one-way; that is, the components only know about the layers below and communicate with only those lower modules; they have no knowledge of what is in the layers above (22:200)(23:72). Examples of a layered architecture are operating systems and data base management systems.

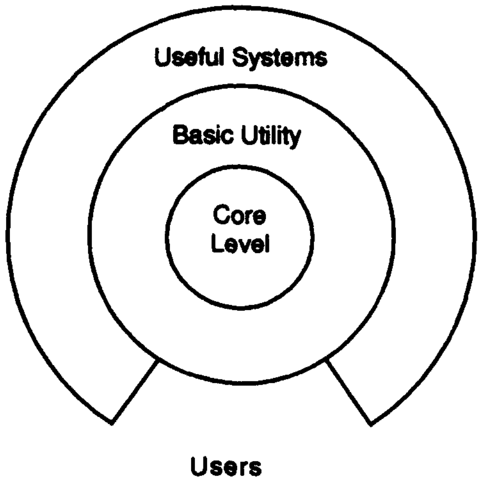


Figure 2.1 Layered Systems(3)

2.2.2 Batch Transformation or Pipe Architecture. A batch transformation architecture is a sequential input-to-output process, in which inputs are supplied at the beginning of the pipe and outputs are presented at the other end of the pipe. Figure 2.2 presents a batch transformation architecture. The architecture is broken down into stages where each stage performs one part of the transformation on the input data. Each stage only knows about the stages immediately preceding it and following it, as well as the inputs and outputs associated with its particular stage. There are no interactions with the outside world once the transformation starts (22:212).

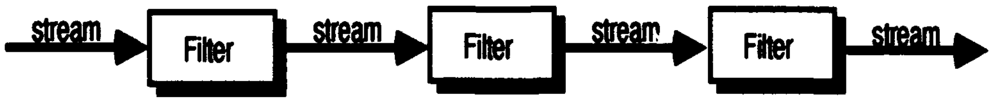


Figure 2.2 Filters and Pipes(3)

2.2.3 Real-Time Architectures. A real-time architecture is used when the timing and performance criteria of a particular system are considered essential to the design; i.e., for any critical actions that the software must perform, the action must be done in an absolute interval of time. Real-time architectures are usually complex and involve interrupt handling, prioritizing tasks, and coordinating resources among multiple processors. Subsequently, their software is often non-logically structured (22:216). An example of a real-time system is a flight control system on any aircraft.

2.2.4 Object-Oriented Architecture. An object-oriented architecture contains models which encapsulate both data and behavior into a single entity (22:1). Models within an object-oriented architecture are representations of real world entities. Each model or object contains attributes and operations; the attributes contain the state information of the object, while the operations are used to change the state of the object. An object-oriented architecture consists of objects changing the state of other objects by invoking the other's operations via message passing.

2.2.5 Production System or Rule-Based Architecture. A production system or rule-based architecture consists of a set of rules, a knowledge base, a control strategy, and a rule applier. Figure 2.3 is a graphical representation of a rule-based system. The set of rules provide a description of the operation to perform when a particular rule is applied. The knowledge base contains information applicable to the task at hand. The control strategy specifies the order in which the rules are compared to the data, and it has a way of deconflicting rules when multiple rules can be applied. Finally, the rule applier applies the rule selected as specified by the control strategy (21:36). An example of a rule-based system is a hospital's diagnostic station.

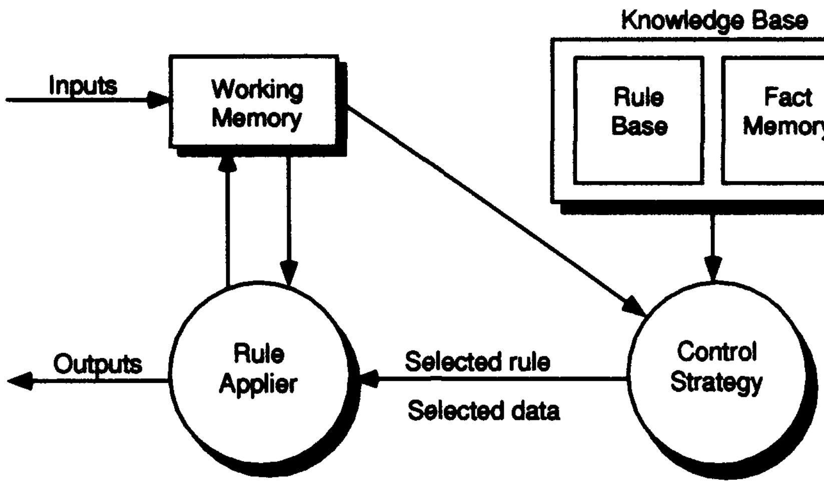


Figure 2.3 Rule-Based Systems(3)

2.2.6 Blackboard Architecture. A blackboard architecture consists of knowledge sources, a blackboard, and a control system. The knowledge sources (ks) are a set of independent modules that contain the system's domain-specific information. The blackboard is a shared data structure through which the knowledge sources communicate with each other (see Figure 2.4). Finally, the control structure determines which knowledge source has access to the blackboard to perform some operations or communications. A blackboard architecture operates by allowing the knowledge sources to post items on the blackboard, read items from the blackboard, or act upon messages posted on the blackboard (21:439).

2.3 Software Architectures Versus Object-Oriented Design

Software architectures have been defined as a style of composing software components; this should not be confused with the term object-oriented architecture. Object-

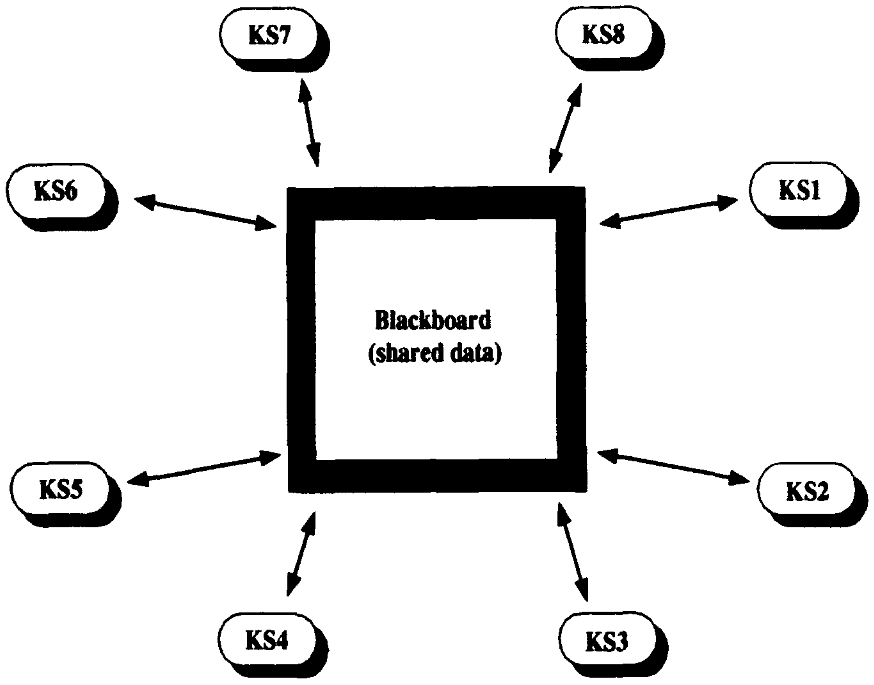


Figure 2.4 Blackboard System(3)

oriented architecture allows the software system designer to "abstract-out" components of a software architecture; this abstraction then allows the system developer to focus on the high-level design issues (22). The use of an object-oriented design in the development of a system's software architecture allows the components of the architecture to be loosely coupled, while providing a high level of cohesion within the component. The loose coupling and high cohesion is achieved through the encapsulation of the behavior and data structure in the software (22:1). To use the pipe architecture as an example, the pipes or filters can be designed as separate entities or objects under the object-oriented paradigm. Data is passed between the filters as previously shown in Figure 2.2, but each filter performs its operations on the data by having one of its methods invoked. This does not violate any principles of a filtering architecture; that being each stage knowing only of its predecessor and its successor.

2.4 Domain-Specific Software Architectures (DSSA)

Before a good definition of DSSA can be given, some additional background terminology is needed to complete an understanding of DSSA, as well as distinguish DSSA from other software architectures.

The first term that needs to be understood is "model". A model, as presented in Section 1.1, is a codified body of knowledge in reusable form (10). This implies that there are some standard interfaces for communicating with and controlling this model. Once the interfaces are standardized, this model is a semi-autonomous set of code that can be used in the composition of larger systems. In other words, the model is reusable.

The important aspect of a model is its codified body of knowledge. This knowledge represents the information needed to accomplish its behavior. Most of the information is local to the model; other information may have to be imported via the communication interface (14).

Models are the basis for DSSAs. The next step in understanding a DSSA is to understand what is meant by domain analysis and domain-specific language. Domain analysis is the generalization of a problem area, where a problem area is a narrow area of interest. Prieto-Diaz defines domain analysis as the process used to identify, capture, and organize information in a domain of interest (19:47-48). As a result of domain analysis, certain entities become evident within the problem area. These entities evolve into models, which are used in the subsequent development of the system.

A domain-specific language is developed as part of a domain analysis. The entities/models formalized from the domain analysis become part of the domain-specific language. The domain-specific language includes any operations, actions or communications between the models of the problem area.

A DSSA includes a software architecture (drawn from the domain analysis), domain models, a domain-specific language, and software engineering knowledge (18). A DSSA defines a way of composing domain models, using a set software architecture to solve a family of related software problems (18). The domain-specific language defines the actions and operations of the domain models within the DSSA. Finally, the software engineering knowledge is captured in both the software architecture and the domain models within the system.

Lowry defines an architecture as a high level description of a generic type of software system: "it includes functional roles of major software components and their interrelationships stated in an application oriented language for use in reasoning and composing prototype components" (16). The preceding sentence describes, in a general sense, an application composition and generation system, which is a transformation system for a problem domain or space (16). Further, a transformation system is a combination of specialized components and software engineering knowledge so that the software is developed, modified, and maintained at the specification level and automatically transformed to implementation or target code (16). This explicitly describes Architect, which is a ongoing research effort (3, 28, 20).

2.5 Example Systems and Their Software Architectures

The following are some software architectures currently being employed or defined by the software industry today.

2.5.1 DSSA for Intelligent Guidance, Navigation and Control. Honeywell and the University of Maryland are currently developing a software architecture for intelligent (adaptive) guidance, navigation and control (1). They are proposing a layered architecture with tools to support the different layers within the architecture. Their architecture will be amenable to automatic analysis, configuration, and population with very heavy reliance on formal methods. Ultimately, they are striving for formal verification in-the-large which will lead to improvements in overall quality in the system being developed. The user will specify source modules to be included in a system, how these modules are to be scheduled, and how these modules are to communicate. The tool set performs the hard real-time scheduling, sensitivity analysis, timing, data, reliability analysis and then automatically assembles the modules into the final system, generating all necessary code.

2.5.2 DSSA for Avionics System. IBM, along with researchers from MIT, Cornell, the University of Texas at Austin, and the University of California at Irvine, is striving to create a workstation-based environment to support the development, maintenance and upgrade of avionics systems with an order-of-magnitude improvement in quality and productivity over current approaches through the reuse of large portions of well-designed and documented software (8). Any system for navigation, guidance, and flight director must manage the routing of source data from suppliers to consumers with minimum handling of this data to prevent "data aging". Their approach to DSSA is to use constraint-based reasoning tools to guide the user, using domain-specific terms, in selecting adaptation values for the system under development. The models are automatically assembled into an executable prototype, which may be run to examine the system behavior. IBM proposes doing a domain analysis of the problem space to identify common concepts. The concepts to be modeled will have well defined interfaces and will include flexible combination mechanisms. IBM believes that these formal interfaces will characterize the semantics of each model(behavior), entry and exit constraints, performance and timing constraints, dependency constraints (layering), and sequencing. Since the architecture is set and provides a structure or solution for a class of problems, the analyst can select and adapt these models for the task at hand using the defined architecture.

2.5.3 VHDL. VHDL (VHSIC (Very High Speed Integrated Circuit) Hardware Description Language) was designed by the U. S. government to standardize the VLSI (Very Large Scale Integration) chip design process and manage the large volumes of data needed in designing a new circuit. VHDL is a standardized design and description language that integrates three different models into one. These interdependent design models are the behavioral model, timing model, and the structural model. VHDL has been designed such that the structural model and behavioral model can be intertwined where their individual boundaries are blurred within an application, or the structural model and behavioral model can be distinct and separate entities within an application. The smallest executable artifact in VHDL is an Entity. An Entity takes signals and internal attributes and generates new signals. Similarly, since an entity may consist of components, further transformations of in-signals are possible (15).

2.5.4 μRapide. μRapide is an executable architecture definition language intended primarily for defining time-sensitive concurrent systems. This language is derived as a subset of the RAPIDE-1 prototyping language which contains object-oriented features and reactive programming constructs. μRapide is an event processing language, where events are tuples of information. The information in an event may include: originator, consumer, time, activity to do, activity done, data values, etc. The semantics of μRapide are based on event processing; that is, generating events, sending events, receiving events, and deciphering events. The authors of μRapide believe this to be the most general way to model communications between components that are computing independently. Subsequently, synchronous and serial communications can be modelled in terms of event processing as well (17).

2.5.5 OCU. The software architecture used by Architect for modeling a system's behavior, is the Software Engineering Institute's (SEI) Object Connection Update (OCU) (14). The major component or level of abstraction within the OCU model is the subsystem. A subsystem consists of objects, an import area, an export area, and a controller. The controller is a self-contained unit; it is the locus of the mission. Subsequently, the objects within the subsystem provide the state of the subsystem. The controller of a subsystem is only aware of subordinate objects for which it has control over; it is unaware of any other superior or sibling subsystems. The only interface a subsystem has with its environment is through the import and export areas.

Due to the way the SEI has defined objects within the OCU model (with standardized interfaces and operations performed by the objects), objects and subsystems can be composed almost arbitrarily. The architecture does not impose any constraints on object or subsystem composition as long as the data interfaces to these components are standardized.

The highest level of abstraction within the OCU model is the Application Executive. The Application Executive controls subsystem activities and scheduling.

2.5.6 Conclusion. My research revolves around enhancing the Architect prototype application composition and generation system as defined by (16). All the software architecture projects presented in Section 2.5 of this chapter are targeting their software for a particular application and domain. At AFIT, the research is focused on developing an application composition and generation capability that transcends many domains. My particular research is focused on uncovering commonalities between software architectures by identifying a suitable framework for comparing different architectures. The commonalities among architectures should eventually lead to more than just code reuse, it could lead to design reuse. Design reuse is the reuse of the knowledge that went into designing a software system or smaller component (architectural fragment) of the system.

The initial implementation of the OCU model in Architect has the domain-specific information of the problem space captured at the primitive level. Most of the higher-level design information has been captured in the software architectural model, OCU. As such, the present software architecture in Architect does not directly incorporate domain-specific information. This provides Architect with the capability to cross domain boundaries.

III. Evaluation of Architect's Software Architecture

3.1 Introduction

Architect is a prototype implementation of an application composition and generation system; as a result, there are certain aspects that still need to be investigated and incorporated into Architect. This research effort focuses on identifying any additional software architectural components required to support the OCU concept of an Application Executive, comparing the OCU software architecture against other software architectures, as well as implementing several already defined enhancements.

3.1.1 Background. The initial implementation of Architect by Anderson, Randour, and Weide (3, 20, 28) produced a limited capability application composition and generation system. For example, Architect could only simulate models that exhibited a sequential mode of behavior (as defined by a subsystem's Update algorithm). Additionally, the system was validated using only two domains, one pedagogical and the other logic circuits, neither of which exhibit any complex behaviors such as feedback.

3.1.2 Current Research Issues.

1. Application Executive: More research is needed to define the application executive role of an application composition and generation system. Welgan is presently defining a domain model for an application executive (29). As part of his domain model validation, the application executive within Architect will be implemented based on the OCU model. The application executive will be a collection of objects composed into a subsystem that provide executive services to the application being modelled.

An additional requirement identified by previous researchers was the incorporation of different modes for simulating an application's behavior. The additional modes of execution all incorporate the processing and management of events. These different modes of execution are addressed in (29); however, defining the structure of events, along with their processing and management within subsystems and primitives is accomplished in this thesis.

2. Incorporation of Additional Domains: Additional domains also need to be modelled. Waggoner and Warner are populating Architect's technology base with new domains; the new domains are missile guidance components and digital signal processing components, respectively (26, 27). Furthermore, Waggoner is also implementing a time delay to the initial digital circuits domain (26).

3.2 Review of Alternative Architectures

To evaluate the OCU architectural model, other architectures were reviewed and analyzed. The architectures are either in use today as a fully functional software set or are being developed. The software architectures reviewed were VHDL (15), MetaH (25), μRapide (17) and IBM ADAGE (8, 6).

3.2.1 Determine a Common Framework. As part of the comparison of the alternative architectures to the OCU model, a framework was needed that would adequately describe all five of the architectures. Identifying a common framework for comparing architectures is recommended by Allen and Garlan in (2:1). This framework needs to allow for the structural analysis of each software architecture, as well as the comparison of the behavior of each of the architectural components within an architecture and among architectures. The framework developed during this research is presented in Chapter IV.

3.2.2 Enhancements Based on Alternative Architecture Analysis. As mentioned in Section 3.1.1, previous research identified a need to incorporate events and event processing within Architect. Since events are not included in the SEI document, the implementation considerations of events within the OCU model should stay within the spirit of the OCU model (14). In other words, event properties incorporated in the OCU architecture in Architect should not violate established properties of the OCU model.

The incorporation of events led to altering the Refine implementation (within Architect) of the OCU architecture model to include events as part of the architecture. Adding events required including event processing and event management within the OCU execution model. The implementation details of event processing and event management are discussed in Chapter VI.

3.2.3 Event Processing. The initial step of incorporating events into the OCU model of Architect was to identify the events processed by a subsystem. Since Architect involves several research efforts (29, 26, 27, 9), a consensus by research members was needed in the identification of events and event types. The consensus resulted in two different classes of events being identified: application events and executive events. As the names of the events imply, the target of each class of event is either the application subsystem being modeled or the executive subsystem of the application executive. Different event scenarios highlighted that all subsystems need to manage (route) all event types; however, the actual processing of events is left to the appropriate subsystem. That is, the executive subsystems process executive events and the application subsystems process application events.

The goal of adding event processing was to enhance Architect's OCU software architecture. This enhancement included defining the structure and information applicable to the application events. As a result, three application event types were identified: Update, SetState, and NewData. The Update event is used to tell a subsystem or primitive to execute. Likewise, the SetState is used to set new attribute values within a primitive to establish its new state. Finally, the NewData event is passed to an independent subsystem by the executive to indicate to the subsystem that it has new data in its import area.

The next step in incorporating events into Architect was to modify the simulation capabilities within the present implementation of the OCU model to handle events. As part of incorporating event processing, we had to preserve the sequential processing capability mentioned in Section 3.1.1. Additionally, not only was the passing of events in the execution of the application important, but the persistence of the events had to be maintained. This persistence increases Architect potential for modelling truly concurrent systems where event causality is important. The persistence requirement implied modification of the subsystems to capture and hold events being sent to the subsystem, as well as capturing events that the subsystem may have originated. Since Architect has a behavior simulation capability, the inclusion of events and event processing imposed changes to the application execution functions within Architect. The original execution programs only allowed for sequential execution of an application as derived by the update algorithm of a subsystem. The new execution programs had to allow for the processing and handling of events within the subsystems. Further, the behavior simulation environment required the addition of some new functions allowing events to be created during the simulation of an application.

With the inclusion of events and an event-driven simulation capability within Architect, the application specialist does not input an execution algorithm as is done with the sequential mode of execution. This forced changes to the semantic checks within Architect. The new semantic checks determined whether an update algorithm within the controller of a subsystem is legal or not.

Introducing events to subsystems is of little use unless the subsystem knows what to do with the event. The controller of a subsystem was enhanced to evaluate events it receives to determine if it needs to process the event or route it to another subordinate subsystem. The event implementation also required the subsystem's controller to know how to actually route the event to any subordinate subsystems. On the other hand, if the controller of a subsystem receives an event that it is to process, it needs to interpret the event and take some action based on the event and event information received. The operational concept of event processing and handling within a subsystem is provided in section 3.4.

3.3 Enhancements As a Result of Adding New Validating Domains

Research involving comparison of the OCU software architectural model to other software architectures may not readily identify all the potential shortcomings within the OCU model. While the comparisons with the other architectures may identify a component that should be part of the OCU model, it may not identify how to properly incorporate the this component into Architect's implementation of the OCU model. Further, with the limited domain analysis of the few problem spaces that are now in Architect's technology base, the new component "may appear" to work. However, as more domains are incorporated into Architect's technology base, it may be discovered that the initial implementation of the architectural component was inadequate. The shortcomings of any new architectural component may only be identified by incorporating new validating domains into the OCU model.

To illustrate the above fact, the initial implementation of the OCU SetState operation within Architect was inadequate for the event-driven simulation capability. A new SetState operation was needed to pass more information than originally required by the sequential model of execution. Additionally, any use of time delays required a primitive's present state to be persistent until a point in time in the future. By modifying the functionality of the OCU SetState operation, we were able overcome simulation rollback problems regarding a primitive's state. This modification allowed a primitive, at a future time (through simulating a delay), to make its new state available.

One of the other research efforts at AFIT is to define a domain model for an application executive (29). As presented in Section 3.1.1, the validation of the application executive domain model will be through the implementation of the OCU-based domain model into Architect. The incorporation of this application executive domain model into Architect forced additional refinements to the current implementation of the OCU model. Mainly, the imports and exports for subsystems required altering. In keeping with SEI's notion of the imports and exports, as well as copying an idea of abstraction from (3, 20)[1],

[1]The consolidation of the imports and exports for a subsystem's subordinate primitive at the subsystem level with pointers back to the owning primitive

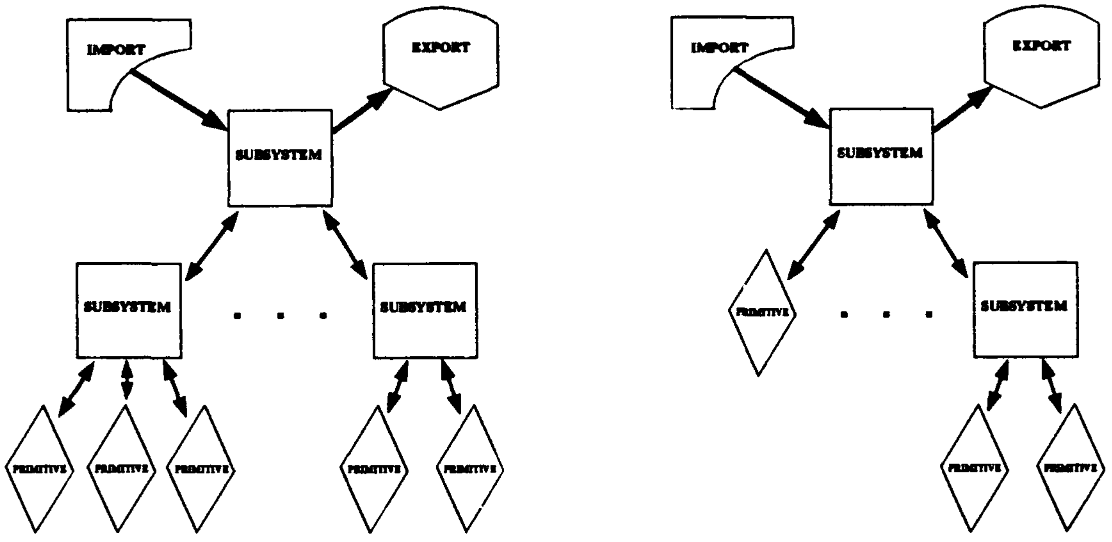


Figure 3.1 Two Independent Subsystems within Architect

the imports and exports were consolidated into an import area and export area of the highest level (superior) subsystem of an independent subsystem. An independent subsystem can be considered a hierarchical structure of subsystems and primitives that is a self-contained composition. Figure 3.1 shows two independent subsystem with imports and exports located at the superior subsystem.

The two subsystems in Figure 3.1 do not exhibit a parent-child relationship, nor do they share the resources of their subordinate objects. Independent subsystems only share information via the import and export areas; furthermore, an independent subsystem is unaware of the existence of any other subsystems. Finally, an independent subsystem exhibits a complete behavior.

Since the inclusion of the application executive required multiply independent subsystems to communicate, there needed to be a central place to find an independent subsystem's imports or exports. This consolidation of imports and exports of an independent subsystem had to keep with the SEI's intention of anonymity of objects (subsystems and primitives). Further, this anonymity had to allow for loose coupling between objects in a composition (12).

3.4 Operational Concept for Event Processing

The application composition process as defined in (3, 20, 28) has not changed with the incorporation of events; changes are restricted to the execute and semantic check requirements. The highlighted boxes in Figure 3.4 emphasize the areas of the overall system design changed to allow the processing and management of events. There are additional changes required to fully implement event processing in Architect; however, the focus of this research was with the changes required at the subsystem and primitive level. The servicing of events by the application environment, event management for a simulation, and the specification of the initial events for an application within Architect is addressed in (29).

With the addition of an event-driven simulation mode, Architect must now query the application specialist during composition for the execution mode of the application. This information is retained as part of the application's definition and made available to other Architect components upon request. A conceptual model of both the non-event-driven mode of execution and event-driven mode of execution within a subsystem can be seen in Figures 3.3 and 3.4, respectively.

After the application specialist finishes composing a new application, he is still required to run a semantic check against the composition. The semantic checks have been changed to check for any Updates in the update algorithm of the controller of an event-

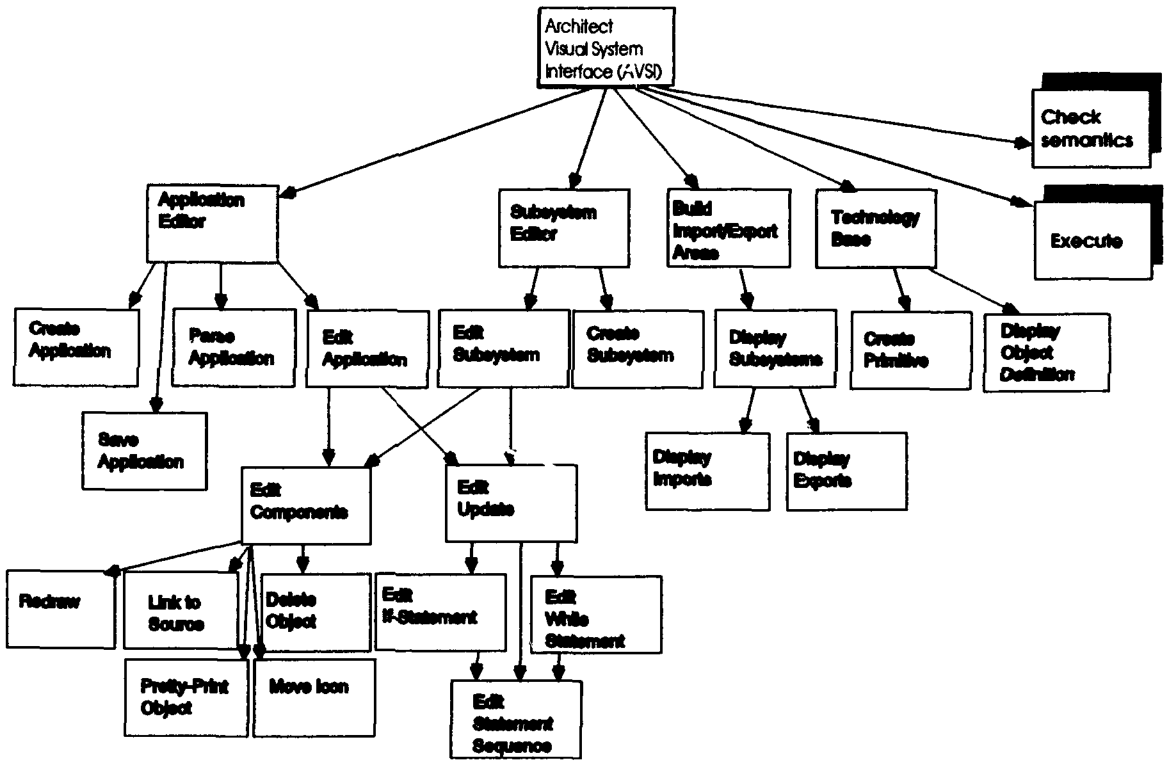


Figure 3.2 Requirement Areas to Change Due to Event Driven Execution

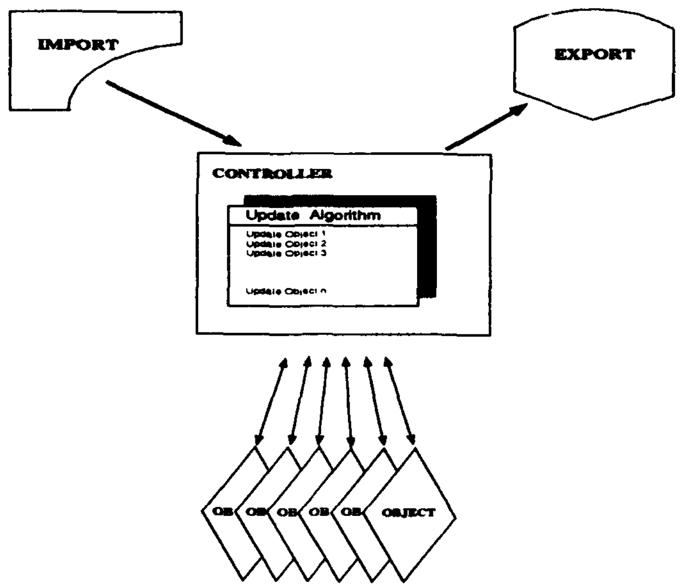


Figure 3.3 Conceptual Model of a Subsystem in the Sequential Mode

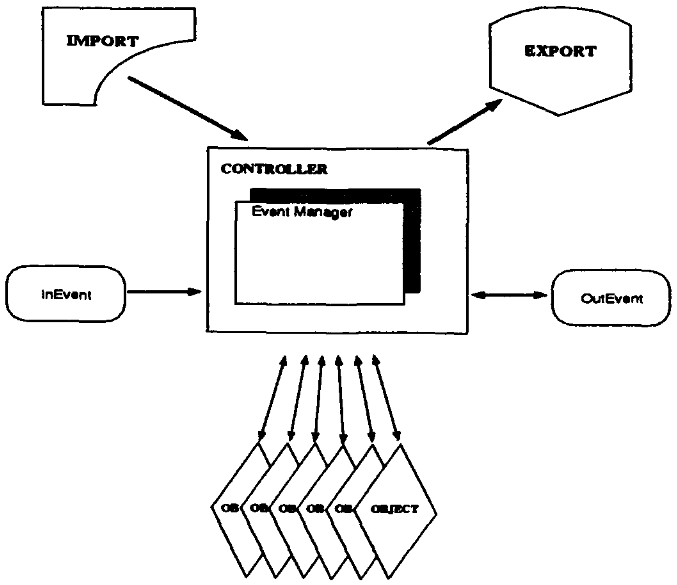


Figure 3.4 Conceptual Model of a Subsystem in the Event-Driven Mode

driven subsystem. Although this is required by the non-event-driven mode of Architect, it is illegal in the event-driven mode.

Once the semantic checks are passed, the application specialist can execute an application. The initial set of events that are needed by Architect to start the execution of an application are predetermined by the application specialist. After composing an application, Architect prompts the application specialist for the initial set of events required for the application in order to initiate the simulation of the application's behavior. The prompting for initial events and their processing is addressed in (29, 9).

After execution starts, the event manager of the application executive selects an event to process. The event manager interprets the event and determines the subsystem that needs to process the event; it then passes the event to the InEvent area of the appropriate independent subsystem. Finally, the application executive passes the flow of control to the independent subsystem that received the event.

Upon receiving flow of control from the executive, the subsystem's controller checks its InEvent area and selects an event from the InEvent area.

The controller interrogates the event and decides what needs to be done next:

- If the event is for this particular subsystem, the controller processes the event and invokes the appropriate subordinate primitive's operation, based on the event type. As an example, if the subsystem receives an Update event for one of its primitives, then the controller will invoke the particular primitive's Update algorithm. When the primitive finishes the operation, any new events generated are passed from the primitive back to the superior subsystem, which returns them to the application executive.

- If the event is not for this particular subsystem but for a subordinate subsystem, the subsystem places the event in the subordinate subsystem's InEvent area and passes the flow of control to the subordinate subsystem.

Subsystems do not do any further processing on events that are forwarded from subordinate objects; subsystems only process events that are received from other invoking subsystems. After a subsystem has processed all events in its InEvent area, it passes all newly generated events in its OutEvent area back to the caller. Finally, before passing flow of control back to the caller, the subsystem clears both the InEvent and OutEvent areas.

3.5 Summary

Architect's initial implementation employed a subset of the OCU model as defined by the SEI. This thesis focuses primarily on defining a framework for comparing software architectures, and determining the suitability of the OCU architecture model for use in application composition and generation systems.

This research also includes expanding the OCU model within Architect to incorporate events and event processing. This chapter discussed the changes to the subsystem and primitive structure and the simulation capability as a result of incorporating events, as well as presented an operational concept for event processing within Architect. Since the initial documentation on the OCU model does not discuss events, the event types and structure of the events had to be identified in addition to any other requirements for event processing and management.

Subsequent chapters of this thesis describe the framework for comparing software architectures, the results of comparing the alternative architectures with the OCU model, the detailed design for events and event-driven simulation within Architect, and the validation of event processing within Architect.

IV. Software Architecture Comparison Methodology

4.1 Introduction

This chapter develops a framework for comparing software architectures. The framework includes a way to compare software architectures both syntactically (structurally) and semantically (behaviorally).

4.2 Framework for Comparing Software Architectures

As mentioned in (3:7-5) and Section 1.2, the OCU model needed to be further evaluated for its suitability as the software architectural model for an application composition and generation system. In an attempt to further evaluate the OCU software architecture model, I chose four other software architectures to evaluate against it. These software architectures are MetaH as defined by Honeywell (25), VHDL as defined by Lipsett (15), μRapide as defined by Luckham and Vera (17), and hierarchical software systems as defined by Batory (6, 8). In order to accurately compare the different architectures, a framework is needed that captures the salient points of each architecture. Furthermore, the framework needed to be capable of being applied to all the architectures being compared.

The methodology for describing an architecture employed what is commonly used to describe any object: a visual description of the object with some description of its behavior. This perspective for evaluating architectures closely correlates to Vestal's attempt to compare architectural description languages (24). He attempted to compare the languages syntactically (structurally) as well as semantically (behaviorally).

4.2.1 The Syntax of the Software Architectures. A method was needed to capture the objects within a software architecture and their relationships. For this purpose, Coad, Yourdon, and Rumbaugh recommend object diagrams (7, 22). The object diagrams for each architecture are organized in such a way to show structural similarities between the architectures, as well as between the objects within an architecture.

4.2.2 The Semantics of the Software Architectures. Structural comparison is not enough to substantially compare the architectures. Just because two objects are structurally the same does not imply they are the same in other respects. For example, a visual comparison of a *Repeat-Until* and a *Do-While* loop reveals that both iterate over a set of program statements a number of times. But closer inspection of the semantics of each loop construct reveals the *Repeat-Until* executes the program statements at least once, where the *Do-While* may not execute any of the statements. We therefore need to examine the behavior (semantics) of the objects to get a detailed understanding of what exactly the object does. This analogy applies to software architectures as well. Not only must we compare the structures of the architectures, but we must compare the semantics (behavior) of the components of the architectures as well.

"Semantics are commonly divided into two classes, static semantics and dynamic semantics" (13). Static semantics require that all components exist and that all components and operands are type-compatible. Dynamic semantics specify what a component does, that is, what it computes. Axiomatic definitions may be used to model execution at a more abstract level than operational models (an operational model explicitly describes a program in terms of changes to a defined state). Furthermore, these definitions are based on formally specified relations or predicates that relate components. The axiomatic approach is good for behavioral correctness because it avoids implementation details and concentrates on how relations among variables are changed by a "program" execution (13). The axiomatic approach of comparing a program's state space is presented in the next section, and it is used to define the semantics of the architectural components of the different architectures.

4.2.2.1 State-Based Machines. The method chosen to compare the semantics of the different architectural components is derived from state-based machines. The choice of the state-based machine approach seems logical since the objects contained in object diagrams encapsulate state information. By using state-based machines, the specific engineering details of the computer (memory size, speed, etc.) and software (looping constructs, array implementations, etc.) are "abstracted out", allowing us to concentrate on the semantics of the architectures independent of any hardware or software considerations (4). By "abstracting out" the engineering data, we can more easily reason about the architectures as well as the subsequent development of the semantics. Ultimately, the semantics are based on formal mathematical concepts and logic, which show program correctness. Before proceeding further, we provide a few definitions to help explain the semantics of state-based machines.

An abstract machine may be represented as $Machine \triangleq (q, F)$ where:

- q is the current state of $Machine$, and
- F is a set of transformations effecting state changes. If Q is the state space of $Machine$, then $F : Q \longrightarrow Q$.
- A state q of a $Machine$ is given by all the data objects O_j within $Machine$

The abstract machine implies the underlying state space may be defined by $Q = Q_1 \times \ldots \times Q_n$ where Q_i is the set of legitimate states of object O_i in *Machine*. Thus, F may represented as:

$(Q_1, Q_2, \ldots, Q_n) \xrightarrow{F} (Q_1, Q_2, \ldots, Q_n)$ where F transforms a state of Q into some new state.

With the above background, the semantics of an architectural component from any of the architectures can be defined axiomatically. The axioms consist of three elements: a set of input assertions, a set of output assertions, and a programming construct P. The programming construct P is used to represent F in the above transformations. If the input assertions are true prior to the execution of the programming construct, and the programming construct terminates, then the output assertions must be true. The input and output assertions characterize the legitimate input and output states for a programming construct. The set of input assertions are called *preconditions*, denoted Φ, the set of output assertions are called *postconditions*, denoted Ψ, and the programming construct is denoted P.

Preconditions define constraints on the input or data that a program is required to handle. Such constraints are bounds on the size of the program state space, assumptions about the input data, bounds on variables, and a description of the structure of the input data. Postconditions characterize exactly what the program must achieve. Postconditions are usually the most important part of a program specification. They define the range of a computation. There are three central roles that postconditions play in program development:

- They provide a precise and formal statement of what a program or function should accomplish. Postconditions describe what a mechanism does with out regard to "how" (11:187).
- They can assist with the constructive development of programs.
- They can be used in the constructive proof of correctness for a program.

If a state machine has a start state q_0 and a set of final states $\{q_{final}\}$ (we will have an element of $\{q_{final}\}$ if the machine terminates), then the overall effect of an abstract machine may be defined as:

$$\vdash \{q_0\}P\{q_{final}\}$$

where P is an abstract program that transforms the the start state q_0 to q_{final}. Using the conditions of Φ and Ψ to characterize the state space the semantics (or partial correctness) of a program P may be described as:

$$\vdash \{\Phi\}P\{\Psi\}^+$$

which is read as: "If the initial state q_0 satisfies condition Φ and the program P terminates, then the final state q_{final} will satisfy the conditions given by Ψ."

The state space of an abstract machine is defined as a tuple of information. As an example, let $Q = (Q_1 \times Q_2 \times Q_3)$ where Q_1 through Q_3 are sets of Natural numbers. A program P can be subdivided into program segments $P = p_1, p_2, \ldots, p_{n-1}, p_n$. These program segments are used to identify smaller transformations of the program, and specific instances of $Q = Q_1 \times \ldots \times Q_n$ can be used as intermediate assertions. Thus, we can transform $\vdash \{q_0\}P\{q_{final}\}$ into:

$$q_0\{p_1\}q_1; q_1\{p_2\}q_2; q_2\{p_3\}q_3; \ldots; q_n\{p_n\}q_{final} \Longleftrightarrow \{q_0\}P\{q_{final}\} \Longleftrightarrow \{\Phi\}P\{\Psi\}^+$$

where $q_0 \ldots q_{final}$ are specific instances of $Q_1 \times \ldots \times Q_n$.

To summarize the information above, the semantics of a set of architectural components within a software architecture can be seen as a set of program transformations P given an initial state satisfying the precondition Φ resulting in a final state satisfying the postconditions Ψ.

4.3 Summary

In this chapter, a framework for comparing software architectures was presented. The framework allows for comparison of both the structure of architectures and the semantics by use of preconditions and postconditions. The next chapter presents the actual comparisons of the five software architectures.

V. Comparison and Analysis of Five Software Architectures

5.1 Introduction

This chapter presents the results of creating an object diagram for each software architecture as well as defining the preconditions and postconditions for each architectural entity of interest. The order of presentation of the architectures is the OCU model, MetaH, VHDL, μRapide, and the IBM DSSA.

The purpose of defining a framework for comparing architectures is to allow for the reuse of design and domain knowledge via transformation techniques. With the similarities identified, architectural components from one model may be transformed into architectural components of another model in a behavior preserving way without knowledge loss.

The semantics of the architectural entities of interest are presented using the nomenclature presented in Chapter 4, $\{\Phi\}P\{\Psi\}^+$, where Φ is the precondition, P is an abstract program and Ψ is the postcondition. The postcondition represents the logical variables that may potentially change as a result of the execution of the abstract program P. Further, to indicate a possible change in state of a logical variable, a Z-like (pronounced Zed) notation is used. The Z notation is in the form of a tick (') or decoration after the component, such as, $attribute'$ (2).

5.2 OCU

The analysis of the OCU model is based on the information presented in (14) and this research. This research is included in the analysis since this research subsumes the addition of events and event processing to the OCU model.

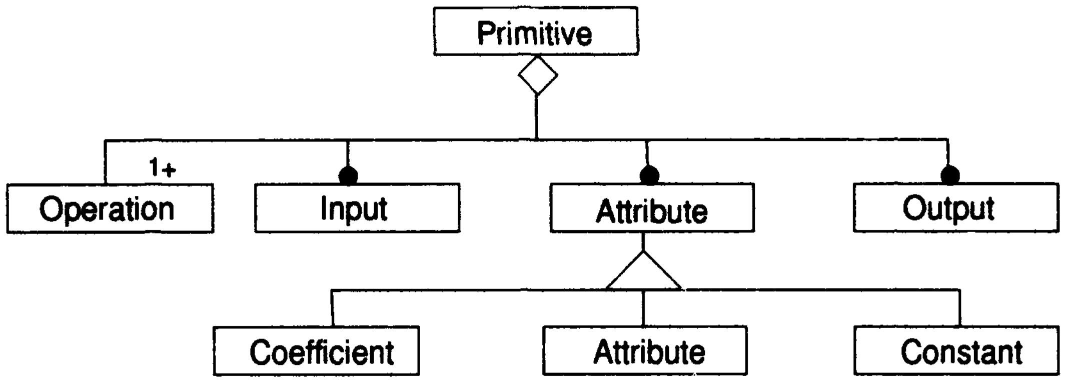


Figure 5.1 Object Model for a Primitive

5.2.1 OCU Syntax. The OCU model consists of primitives, subsystems and an application executive. The lowest level of abstraction is the primitive. A primitive has inputs, outputs, operations, and attributes. The inputs and outputs allow information to enter and exit the primitive. The attributes hold the local state information of a primitive. Finally, the execution of the operations allow the primitive to change state. Figure 5.1 shows an object diagram of a primitive.

The next level of abstraction in the OCU model is the subsystem. A subsystem is composed of imports and exports, inevents and outevents, a controller, as well as subordinate primitives and subsystems. The subsystem manages its subordinate objects, its inevent and outevent areas, and the imports and exports of its subordinate objects. The management of subordinate objects is done via the controller, which has either an update algorithm or an event manager, both of which update the state of the subsystem by updating the state of a subsystem's subordinate objects. The update algorithm contains a sequence of object updates (as well as other operations) that must be performed in order for the subsystem to achieve a new state. In contrast, the event manager processes inbound

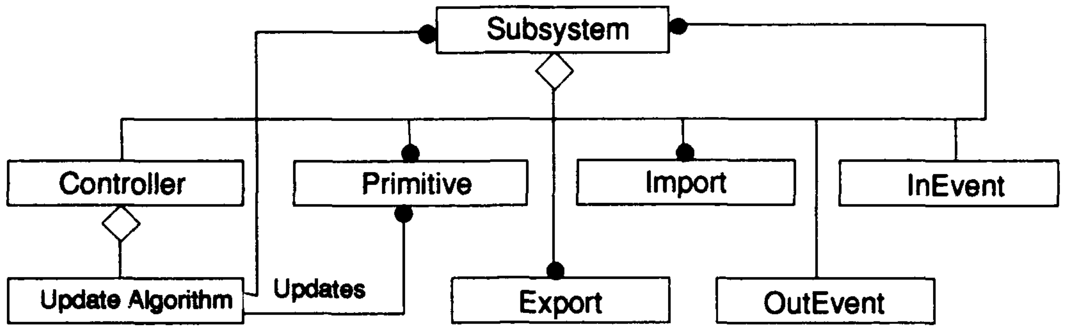


Figure 5.2 Object Model for a Subsystem

events and handles outbound events in order to achieve a new state for the subsystem. Figure 5.2 is the object model for a subsystem.

Finally, the highest level of abstraction in the OCU model is the application executive. The application executive is a collection of primitives and subsystems used to support execution of the behavior of a composed application. Figure 5.3 is a composite object model showing all the components of the OCU architecture together. There is no significance to the shaded, dashed and solid type lines, they only serve to help differentiate the different relationships in the object diagram.

5.2.2 OCU Semantics. Normally, we only analyze the behavior of objects that exhibit noteworthy actions (22:112). Objects that exhibit interesting behavior within the OCU model are the primitive and subsystem objects. These are the architectural entities whose semantics are analyzed.

5.2.2.1 Primitive. A primitive's semantics can be seen in Figure 5.4. Note that program P is composed of the Operations identified in the box. Φ is represented as the following:

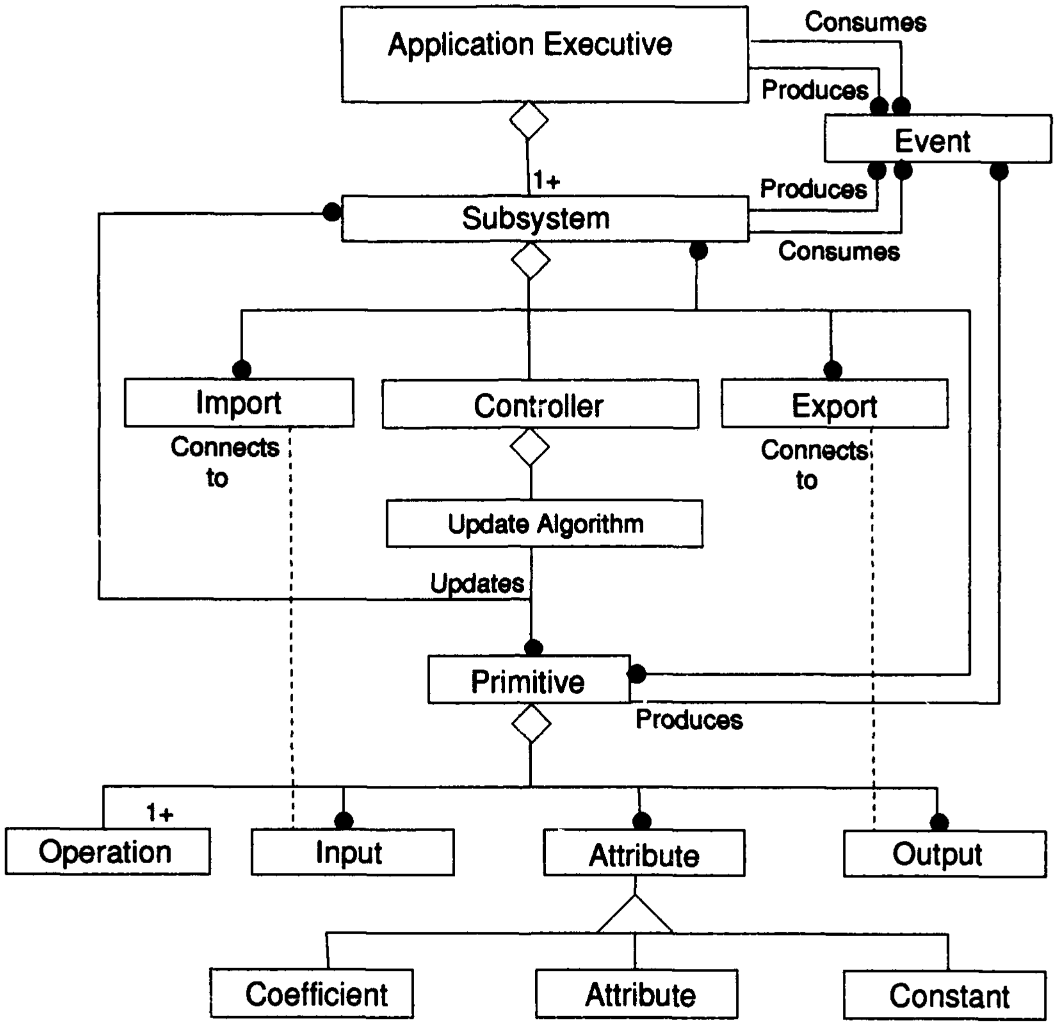
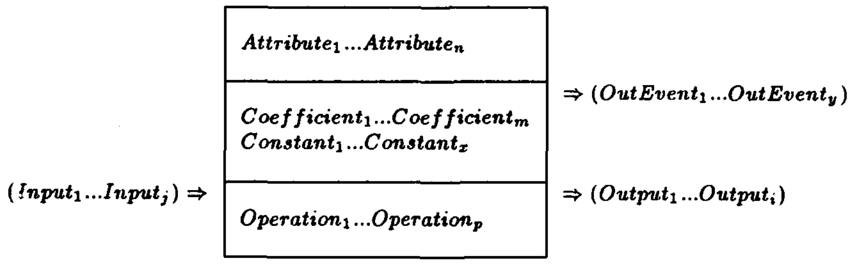


Figure 5.3 Complete OCU Object Model



$Attribute_1 ... Attribute_n$
$Coefficient_1 ... Coefficient_m$
$Constant_1 ... Constant_x$
$Operation_1 ... Operation_p$
$(Input_1 ... Input_j) \Rightarrow$
$\Rightarrow (OutEvent_1 ... OutEvent_y)$
$\Rightarrow (Output_1 ... Output_i)$

Figure 5.4 Diagram of an OCU Primitive Behavior

$$(ValidInput(Input_1) \wedge \ldots \wedge ValidInput(Input_j)) \wedge \{(Attribute_1, \ldots, Attribute_n),$$
$$(Coefficient_1, \ldots, Coefficient_m), (Constant_1, \ldots, Constant_x)\} \in ValidStateSpace$$

"ValidInput" is a predicate that evaluates an Input and ensures that the input value being passed to the Primitive is of a type the primitive can input and within bounds. Additionally, "ValidStateSpace" is the cross product of all the legitimate state spaces that a primitive can occupy based on its Attribute, Coefficient, and Constant values. The precondition above requires that the Input values are of the correct type and range, and the values of the Attributes, Coefficients and Constants together form a valid state space for the primitive.

Two different postconditions are presented for the transformation process P on a primitive. The first Ψ for a primitive represents the the transformation as a result of applying the SetState and Update for the general OCU model.

1. If P represents a SetState operation then

   $$(Attribute'_i = P(Attribute_i)) \quad \vee \quad (Coefficient'_i = P(Coefficient_i))$$

2. If P represents an Update operation then

   $$(Attribute'_1, \ldots, Attribute'_n) = P((Attribute_1, \ldots, Attribute_n), (Input_1, \ldots, Input_j),$$
   $$(Coefficient_1, \ldots, Coefficient_m), (Constant_1, \ldots, Constant_x)) \quad \wedge$$
   $$(Input'_1, \ldots, Input'_j) = P((Attribute_1, \ldots, Attribute_n), (Input_1, \ldots, Input_j),$$
   $$(Coefficient_1, \ldots, Coefficient_m), (Constant_1, \ldots, Constant_x)) \quad \wedge$$
   $$(Output'_1, \ldots, Output'_i) = P((Attribute_1, \ldots, Attribute_n), (Input_1, \ldots, Input_j),$$
   $$(Coefficient_1, \ldots, Coefficient_m), (Constant_1, \ldots, Constant_x))$$

The i subscript in a postcondition uniquely identifies an Attribute or Coefficient that is to change. The postcondition of a non-event-driven execution implies that as a result of the transformation process of P (P representing a SetState operation), an Attribute or a Coefficient value has changed. Likewise, the postcondition (where P is an Update operation) indicates that the Attribute, Input, and Output values have changed.

The second Ψ represents the transformation of a primitive based on our implementation of the OCU model in Architect. Due to event-driven simulation, this transformation differs from the one above.

1. If P represents a SetState operation then

$$\begin{aligned}
&(Attribute'_1,\ldots,Attribute'_n) = P((Attribute_1,\ldots,Attribute_n),(Input_1,\ldots,Input_j),\\
&\quad(Coefficient_1,\ldots,Coefficient_m),(Constant_1,\ldots,Constant_x)) \quad \wedge\\
&(OutEvent'_1,\ldots,OutEvent'_y) = P((Attribute_1,\ldots,Attribute_n),(Input_1,\ldots,Input_j),\\
&\quad(Coefficient_1,\ldots,Coefficient_m),(Constant_1,\ldots,Constant_x)) \quad \wedge\\
&(Input'_1,\ldots,Input'_j) = P((Attribute_1,\ldots,Attribute_n),(Input_1,\ldots,Input_j),\\
&\quad(Coefficient_1,\ldots,Coefficient_m),(Constant_1,\ldots,Constant_x)) \quad \wedge\\
&(Output'_1,\ldots,Output'_i) = P((Attribute_1,\ldots,Attribute_n),(Input_1,\ldots,Input_j),\\
&\quad(Coefficient_1,\ldots,Coefficient_m),(Constant_1,\ldots,Constant_x))
\end{aligned}$$

2. If P represents an Update operation then

$$\begin{aligned}
&(OutEvent'_1,\ldots,OutEvent'_y) = P((Attribute_1,\ldots,Attribute_n),(Input_1,\ldots,Input_j),\\
&\quad(Coefficient_1,\ldots,Coefficient_m),(Constant_1,\ldots,Constant_x))
\end{aligned}$$

The postcondition implies that as a result of the transformation process of P (P representing a SetState operation), the Input, Attribute, OutEvent and Output values change. Similarly, the postcondition of the transformation process (where P represents an Update operation) is the generation of some OutEvents. Changing any of these values dictates that a primitive has changed state.

5.2.2.2 Subsystem. The subsystem's semantics can be seen in Figure 5.5. The state of an OCU Subsystem is a composition of all the state information

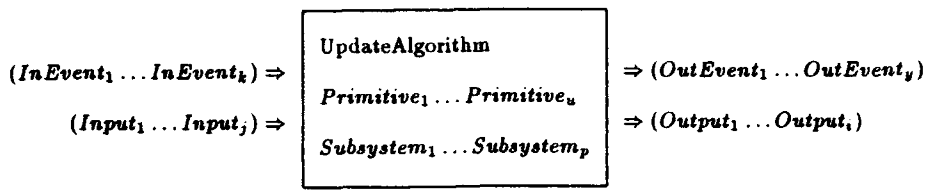

$(InEvent_1 \ldots InEvent_k) \Rightarrow$
$(Input_1 \ldots Input_j) \Rightarrow$
UpdateAlgorithm
$Primitive_1 \ldots Primitive_u$
$Subsystem_1 \ldots Subsystem_p$
$\Rightarrow (OutEvent_1 \ldots OutEvent_y)$
$\Rightarrow (Output_1 \ldots Output_i)$

Figure 5.5 Behavioral Model for an OCU Subsystem With Subsystems and Primitives

of its subordinates. As such, when a subordinate object to a subsystem changes state, the subsystem itself has changed state. Note that program P represents either the sequential execution of the UpdateAlgorithm or the execution of some operation of a subordinate object based on an event received. Φ can be represented as the following:

$$\begin{aligned}&(ValidInput(Input_1) \wedge \ldots \wedge ValidInput(Input_j)) \wedge\\ &\quad (ValidEvent(InEvent_1) \wedge \ldots \wedge ValidEvent(InEvent_k)) \wedge\\ &\quad \{(Primitive_1, \ldots, Primitive_u), (Subsystem_1, \ldots, Subsystem_p)\} \in ValidStateSpace\end{aligned}$$

"ValidEvent" is a predicate that evaluates an InEvent to ensure the target of the inbound event is this subsystem. Also, "ValidStateSpace" represents the cross product of all the legitimate state spaces that a subsystem can occupy based on the state space of its subordinate Primitives and Subsystems. The precondition above requires that the Input values are of the correct type and within bounds, the InEvents are for this subsystem, and the state space of the subordinate Primitives and Subsystems together form a valid state space for the subsystem.

In the non-event-driven transformation process for the general OCU model, P represents the sequential execution of the operations in the UpdateAlgorithm. Ψ is:

$$\begin{aligned}&(Primitive'_1, \ldots, Primitive'_u) = P((Primitive_1, \ldots, Primitive_u), (Input_1, \ldots, Input_j),\\ &\quad (Subsystem_1, \ldots, Subsystem_p)) \quad \wedge\\ &(Subsystem'_1, \ldots, Subsystem'_p) = P((Primitive_1, \ldots, Primitive_u), (Input_1, \ldots, Input_j),\\ &\quad (Subsystem_1, \ldots, Subsystem_p)) \quad \wedge\end{aligned}$$

$$(Input'_1, \ldots, Input'_j) = P((Primitive_1, \ldots, Primitive_u), (Input_1, \ldots, Input_j), (Subsystem_1, \ldots, Subsystem_p)) \quad \wedge$$
$$(Output'_1, \ldots, Output'_i) = P((Primitive_1, \ldots, Primitive_u), (Input_1, \ldots, Input_j), (Subsystem_1, \ldots, Subsystem_p))$$

The postcondition of a subsystem for a non-event-driven execution implies that the subsystem as a whole has changed state, including subordinate Primitives and Subsystems. The Input and Output values of the subsystem have also changed. The reason the Input values can change as a result of the transformation process is due to the possibility of feedback.

With the introduction of events to the OCU model of Architect, a subsystem exhibits a different behavior. After an event-driven transformation process of P, Ψ is:

1. If P represents a SetState event for a subordinate object then

   $$((Primitive'_i = P(Primitive_i)) \vee (Subsystem'_i = P(Subsystem_i))) \quad \wedge$$
   $$((OutEvent'_1, \ldots, OutEvent'_y) = P(Primitive_i) \vee P(Subsystem_i)) \quad \wedge$$
   $$((Input'_1, \ldots, Input'_j) = P(Primitive_i) \vee P(Subsystem_i)) \quad \wedge$$
   $$((Output'_1, \ldots, Output'_i) = P(Primitive_i) \vee P(Subsystem_i))$$

2. If P represents an Update event for a subordinate object then

   $$(OutEvent'_1, \ldots, OutEvent'_y) = P(Primitive_i) \vee P(Subsystem_i)$$

The i subscript in the postcondition uniquely identifies a particular Primitive or Subsystem that is the target of the event. The postcondition for the transformation process (where P is a SetState event) dictates that the uniquely identified subordinate Subsystem or Primitive changes state, as well as some OutEvent, Output, and Input values. The postcondition of the transformation process P for an Update event indicates that only the

OutEvents of a subsystem change as a result of the event being processed by some uniquely identified subordinate Subsystem or Primitive.

5.3 MetaH

5.3.1 MetaH Syntax. A MetaH architecture is specified using eight different objects. The most primitive of these entities is a subgroup called source entities; they are port types, subprograms, packages, and monitors. Source entities have links to software source modules. The next higher-level abstraction is a process; a process entity is a grouping of source entities. As processes are groupings of source entities, macros and modes are higher-level groupings of processes. Likewise, processes, macros and modes can be grouped to form a higher-level mode or macro abstraction. Finally, the highest-level abstraction is an application. The application contains all the information needed to analyze a design against real-time constraints, as well as actually generate code for a target system. An application even contains a hardware description file for the target system, which is used during actual code generation. In order to specialize source modules, MetaH entities contain local attributes. Figures 5.6, 5.7, and 5.8 are object models for MetaH source entities, MetaH higher-level objects and a composite MetaH model, respectively. The shaded, dashed and solid type lines signify the different relationships in the object model.

5.3.2 MetaH Semantics. Objects that exhibit interesting behavior within MetaH are the source entity, process, macro, and mode objects. These are the architectural entities whose semantics are analyzed.

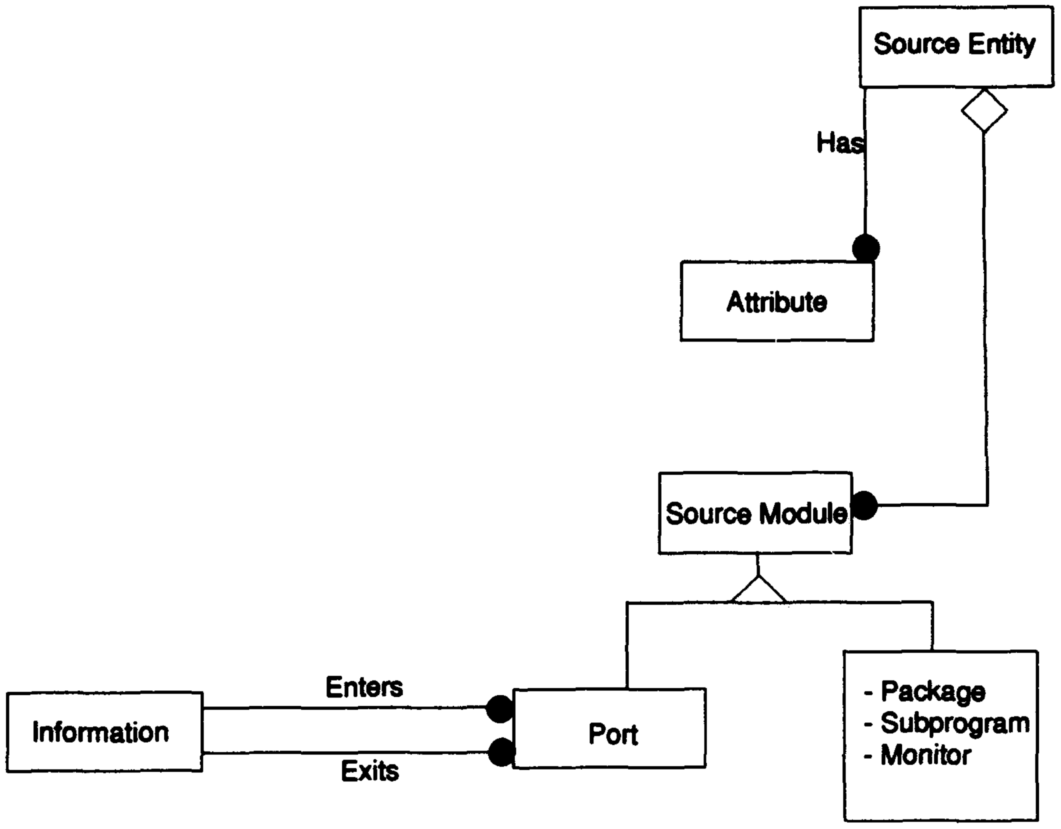


Figure 5.6 MetaH Source Entity Object Model

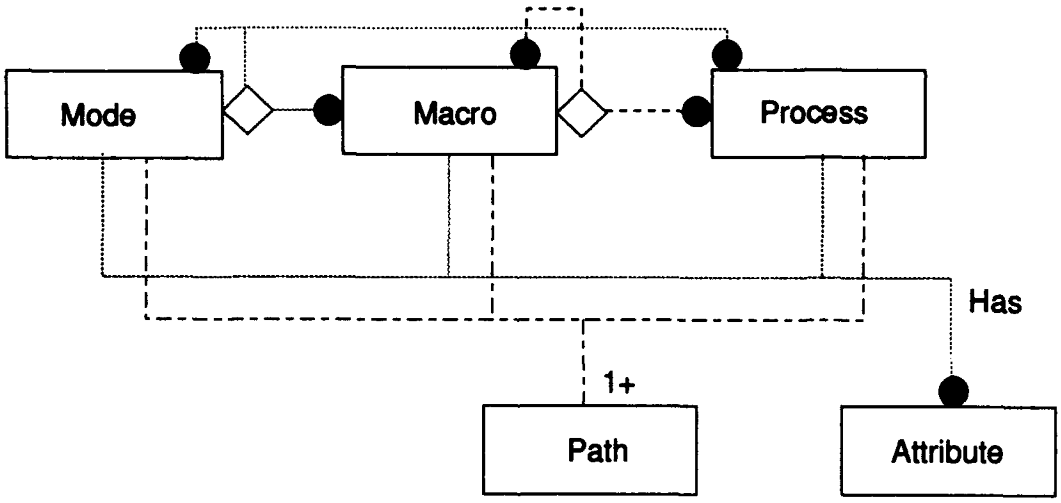


Figure 5.7 MetaH Higher-Level Abstractions Object Model

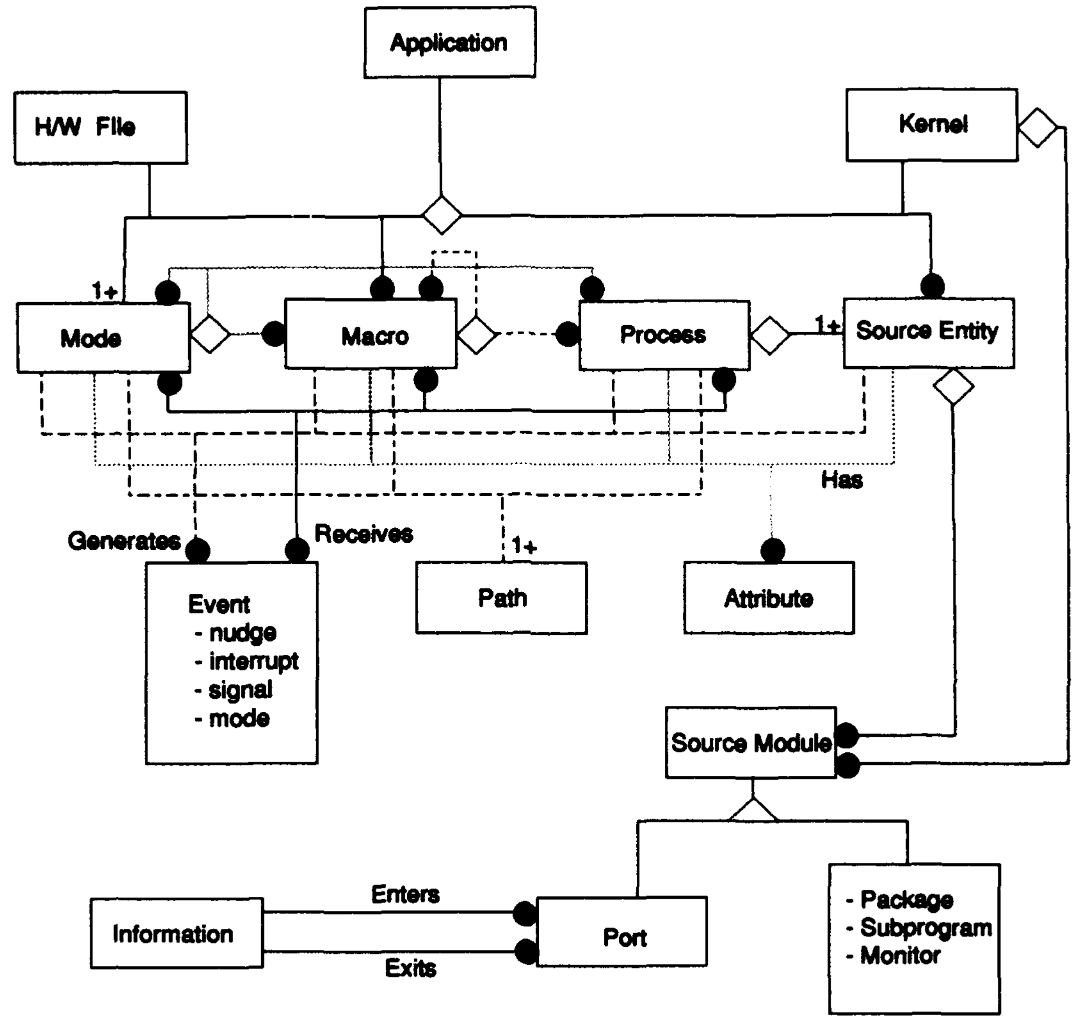


Figure 5.8 MetaH Complete Object Model

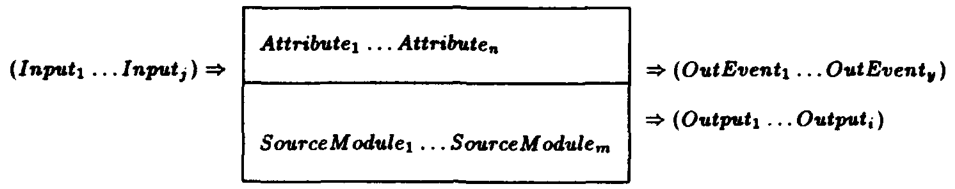

$(Input_1 \ldots Input_j) \Rightarrow$
$Attribute_1 \ldots Attribute_n$
$SourceModule_1 \ldots SourceModule_m$
$\Rightarrow (OutEvent_1 \ldots OutEvent_y)$
$\Rightarrow (Output_1 \ldots Output_i)$

Figure 5.9 Behavior of a MetaH Source Entity

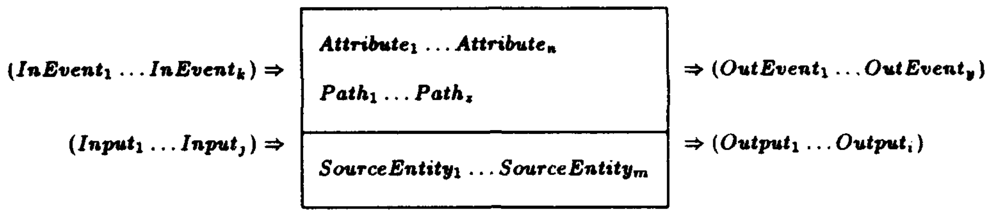

$(InEvent_1 \ldots InEvent_k) \Rightarrow$
$(Input_1 \ldots Input_j) \Rightarrow$
$Attribute_1 \ldots Attribute_n$
$Path_1 \ldots Path_z$
$SourceEntity_1 \ldots SourceEntity_m$
$\Rightarrow (OutEvent_1 \ldots OutEvent_y)$
$\Rightarrow (Output_1 \ldots Output_i)$

Figure 5.10 Behavior for a MetaH Process

5.3.2.1 Source Entity. A source entity's semantics can be seen in Figure 5.9. Note that program P represents any subset of SourceModules identified in the box. Φ can be represented as the following:

$$(ValidInput(Input_1) \wedge \ldots \wedge ValidInput(Input_j)) \wedge \{(Attribute_1, \ldots, Attribute_n)\} \\ \in ValidStateSpace$$

"ValidInput" for the MetaH model represents a predicate that evaluates an Input and ensures that the input value being passed to the source entity is of a type the source entity can input and within bounds. Also, "ValidStateSpace" represents the cross product of all the legitimate state spaces that a source entity can occupy based on its Attribute values. The precondition above requires that the Input values are of the correct type and range, and the values of the Attributes form a valid state space for the source entity.

As a result of the transformation process of P, Ψ will be the following:

$$(Input'_1, \ldots, Input'_j) = P((Attribute_1, \ldots, Attribute_n), (Input_1, \ldots, Input_j)) \quad \wedge \\ (Output'_1, \ldots, Output'_i) = P((Attribute_1, \ldots, Attribute_n), (Input_1, \ldots, Input_j)) \quad \wedge \\ (OutEvent'_1, \ldots, OutEvent'_y) = P((Attribute_1, \ldots, Attribute_n), (Input_1, \ldots, Input_j))$$

The postcondition indicates that as a result of the transformation process of P, the Input, Attribute, OutEvent and Output values change for a source entity. The changing of the Input values are due to the possibility of feedback.

5.3.2.2 Process. Process semantics can be seen in Figure 5.10. The state of a MetaH process is a composite of its Attribute values and the state information of its subordinate source entities. Note that program P represents the execution of some Path. Φ for a process can be represented as the following:

$$(ValidInput(Input_1) \wedge \ldots \wedge ValidInput(Input_j)) \wedge \\ (ValidEvent(InEvent_1) \wedge \ldots \wedge ValidEvent(InEvent_k)) \wedge \{(Attribute_1, \ldots, Attribute_n), \\ (SourceEntity_1, \ldots, SourceEntity_m), (Path_1, \ldots, Path_z)\} \in ValidStateSpace$$

The"ValidEvent" is a predicate that evaluates an InEvent to ensure the target of the inbound event is this Process. "ValidStateSpace" represents the cross product of all the legitimate state spaces that a process can occupy based on its Attribute values, subordinate SourceEntities, and Paths of execution. The precondition above requires that the Input values are of the correct type and within bounds, the InEvents are for the Process, and the values of the Attributes along with the state space of its SourceEntities as constrained by Paths define a valid state space for a process.

As a result of the transformation process of P, Ψ will be the following:

$$(SourceEntity'_1, \ldots, SourceEntity'_m) = P((Attribute_1, \ldots, Attribute_n), \\ (Input_1, \ldots, Input_j), (InEvent_1, \ldots, InEvent_k), \\ (SourceEntity_1, \ldots, SourceEntity_m)) \quad \wedge$$

$$(Attribute'_1, \ldots, Attribute'_n) = P((Attribute_1, \ldots, Attribute_n), (Input_1, \ldots, Input_j), \\ (InEvent_1, \ldots, InEvent_k), (SourceEntity_1, \ldots, SourceEntity_m)) \quad \wedge$$

$$(Input'_1, \ldots, Input'_j) = P((Attribute_1, \ldots, Attribute_n), (Input_1, \ldots, Input_j), \\ (InEvent_1, \ldots, InEvent_k), (SourceEntity_1, \ldots, SourceEntity_m)) \quad \wedge$$

$$(InEvent'_1, \ldots, InEvent'_k) = P((Attribute_1, \ldots, Attribute_n), (Input_1, \ldots, Input_j), \\ (InEvent_1, \ldots, InEvent_k), (SourceEntity_1, \ldots, SourceEntity_m)) \quad \wedge$$

$$(Output'_1, \ldots, Output'_i) = P((Attribute_1, \ldots, Attribute_n), (Input_1, \ldots, Input_j), \\ (InEvent_1, \ldots, InEvent_k), (SourceEntity_1, \ldots, SourceEntity_m)) \quad \wedge$$

$$(OutEvent'_1, \ldots, OutEvent'_y) = P((Attribute_1, \ldots, Attribute_n), (Input_1, \ldots, Input_j), \\ (InEvent_1, \ldots, InEvent_k), (SourceEntity_1, \ldots, SourceEntity_m))$$

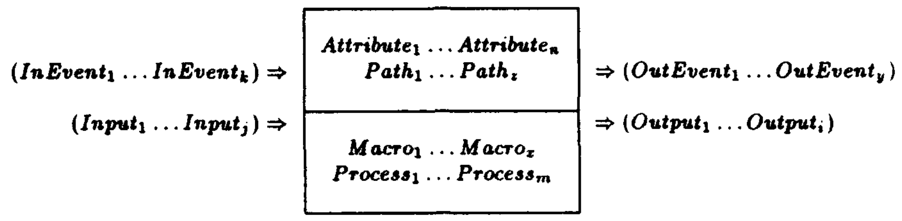

$(InEvent_1 \ldots InEvent_k) \Rightarrow$
$(Input_1 \ldots Input_j) \Rightarrow$
$Attribute_1 \ldots Attribute_n$
$Path_1 \ldots Path_z$
$Macro_1 \ldots Macro_x$
$Process_1 \ldots Process_m$
$\Rightarrow (OutEvent_1 \ldots OutEvent_y)$
$\Rightarrow (Output_1 \ldots Output_i)$

Figure 5.11 Behavior for a MetaH Macro

The postcondition states that as a result of the transformation process of P, the Input, InEvent, Attribute, OutEvent and Output values have changed. Also the state of the process' subordinate SourceEntities have changed state. Changing in the Input values and InEvents are due to feedback.

5.3.2.3 Macro. The macro's semantics can be seen in Figure 5.11. As with a process, a macro's state is a composite of its Attribute values and the state space of its subordinate objects. Program P represents the execution of some Path. Φ can be represented as the following:

$$\begin{aligned}&(ValidInput(Input_1) \wedge \ldots \wedge ValidInput(Input_j)) \wedge \\ &\quad (ValidEvent(InEvent_1) \wedge \ldots \wedge ValidEvent(InEvent_k)) \wedge \{(Attribute_1, \ldots, Attribute_n), \\ &\quad (Process_1, \ldots, Process_m), (Macro_1, \ldots, Macro_x), \\ &\quad (Path_1 \ldots Path_z)\} \in ValidStateSpace\end{aligned}$$

"ValidStateSpace" for a macro is the cross product of all the legitimate state spaces that a macro can occupy based on the values of its Attributes, the state space of its subordinate Macros and Processes, as restricted by the execution Paths. The above precondition dictates that the InEvents received are for the macro, the Inputs are of the correct type and range, and the state space for a macro is defined by its Attribute values, Macros, Processes, and execution Paths.

After the transformation process of P, Ψ will be:

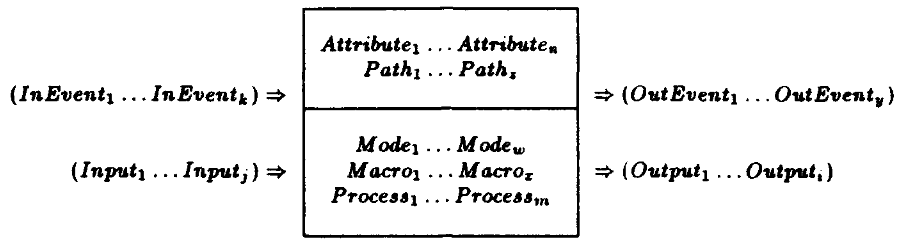

$(InEvent_1 \ldots InEvent_k) \Rightarrow$
$(Input_1 \ldots Input_j) \Rightarrow$
$Attribute_1 \ldots Attribute_n$
$Path_1 \ldots Path_s$
$Mode_1 \ldots Mode_w$
$Macro_1 \ldots Macro_x$
$Process_1 \ldots Process_m$
$\Rightarrow (OutEvent_1 \ldots OutEvent_y)$
$\Rightarrow (Output_1 \ldots Output_i)$

Figure 5.12 Behavior for a MetaH Mode

$$\begin{aligned}
&(Process'_1, \ldots, Process'_m) = P((Attribute_1, \ldots, Attribute_n), (Input_1, \ldots, Input_j),\\
&\quad (InEvent_1, \ldots, InEvent_k), (Process_1, \ldots, Process_m), (Macro_1, \ldots, Macro_x)) \quad \wedge\\
&(Macro'_1, \ldots, Macro'_x) = P((Attribute_1, \ldots, Attribute_n), (Input_1, \ldots, Input_j),\\
&\quad (InEvent_1, \ldots, InEvent_k), (Process_1, \ldots, Process_m), (Macro_1, \ldots, Macro_x)) \quad \wedge\\
&(Attribute'_1, \ldots, Attribute'_n) = P((Attribute_1, \ldots, Attribute_n), (Input_1, \ldots, Input_j),\\
&\quad (InEvent_1, \ldots, InEvent_k), (Process_1, \ldots, Process_m), (Macro_1, \ldots, Macro_x)) \quad \wedge\\
&(Input'_1, \ldots, Input'_j) = P((Attribute_1, \ldots, Attribute_n), (Input_1, \ldots, Input_j),\\
&\quad (InEvent_1, \ldots, InEvent_k), (Process_1, \ldots, Process_m), (Macro_1, \ldots, Macro_x)) \quad \wedge\\
&(InEvent'_1, \ldots, InEvent'_k) = P((Attribute_1, \ldots, Attribute_n), (Input_1, \ldots, Input_j),\\
&\quad (InEvent_1, \ldots, InEvent_k), (Process_1, \ldots, Process_m), (Macro_1, \ldots, Macro_x)) \quad \wedge\\
&(Output'_1, \ldots, Output'_i) = P((Attribute_1, \ldots, Attribute_n), (Input_1, \ldots, Input_j),\\
&\quad (InEvent_1, \ldots, InEvent_k), (Process_1, \ldots, Process_m), (Macro_1, \ldots, Macro_x)) \quad \wedge\\
&(OutEvent'_1, \ldots, OutEvent'_y) = P((Attribute_1, \ldots, Attribute_n), (Input_1, \ldots, Input_j),\\
&\quad (InEvent_1, \ldots, InEvent_k), (Process_1, \ldots, Process_m), (Macro_1, \ldots, Macro_x))
\end{aligned}$$

The postcondition of a macro implies that the macro as a whole has changed state, including its subordinate Processes and Macros. Also changed as a result of the transformation process P are the Attribute, Input and Output values, and the InEvents and OutEvents of the macro. Feedback is the cause for the InEvents and Input values changing as a result of the transformation process.

5.3.2.4 Mode. The mode's semantics can be seen in Figure 5.12. As with a macro, a mode's state is a composite of its Attribute values and the state space of its subordinate objects. Note that program P in mode must represent the execution of some Path from the figure. Φ can be represented as the following list:

$(ValidInput(Input_1) \wedge \ldots \wedge ValidInput(Input_j)) \wedge$
$(ValidEvent(InEvent_1) \wedge \ldots \wedge ValidEvent(InEvent_k)) \wedge \{(Attribute_1, \ldots, Attribute_n),$
$(Process_1, \ldots, Process_m), (Macro_1, \ldots, Macro_x), (Mode_1, \ldots, Mode_w),$
$(Path_1, \ldots, Path_z)\} \in ValidStateSpace$

"ValidStateSpace" for a mode is the cross product of all the legitimate state spaces that a mode can occupy based on the values of its Attributes, the state space of its subordinate objects, as restricted by the execution Paths. The above precondition requires that the InEvents received are for the mode, the Inputs are of the correct type and range, and the state space for a mode is defined by its Attribute values, Modes, Macros, Processes, and Paths.

After the transformation process of P, Ψ will be:

$(Process'_1, \ldots, Process'_m) = P((Attribute_1, \ldots, Attribute_n), (Input_1, \ldots, Input_j),$
$(InEvent_1, \ldots, InEvent_k), (Process_1, \ldots, Process_m), (Macro_1, \ldots, Macro_x),$
$(Mode_1, \ldots, Mode_w)) \quad \wedge$

$(Macro'_1, \ldots, Macro'_x) = P((Attribute_1, \ldots, Attribute_n), (Input_1, \ldots, Input_j),$
$(InEvent_1, \ldots, InEvent_k), (Process_1, \ldots, Process_m), (Macro_1, \ldots, Macro_x),$
$(Mode_1, \ldots, Mode_w)) \quad \wedge$

$(Mode'_1, \ldots, Macro'_w) = P((Attribute_1, \ldots, Attribute_n), (Input_1, \ldots, Input_j),$
$(InEvent_1, \ldots, InEvent_k), (Process_1, \ldots, Process_m), (Macro_1, \ldots, Macro_x),$
$(Mode_1, \ldots, Mode_w)) \quad \wedge$

$(Attribute'_1, \ldots, Attribute'_n) = P((Attribute_1, \ldots, Attribute_n), (Input_1, \ldots, Input_j),$
$(InEvent_1, \ldots, InEvent_k), (Process_1, \ldots, Process_m), (Macro_1, \ldots, Macro_x),$
$(Mode_1, \ldots, Mode_w)) \quad \wedge$

$(Input'_1, \ldots, Input'_j) = P((Attribute_1, \ldots, Attribute_n), (Input_1, \ldots, Input_j),$
$(InEvent_1, \ldots, InEvent_k), (Process_1, \ldots, Process_m), (Macro_1, \ldots, Macro_x),$
$(Mode_1, \ldots, Mode_w)) \quad \wedge$

$(InEvent'_1, \ldots, InEvent'_k) = P((Attribute_1, \ldots, Attribute_n), (Input_1, \ldots, Input_j),$
$(InEvent_1, \ldots, InEvent_k), (Process_1, \ldots, Process_m), (Macro_1, \ldots, Macro_x),$
$(Mode_1, \ldots, Mode_w)) \quad \wedge$

$(Output'_1, \ldots, Output'_i) = P((Attribute_1, \ldots, Attribute_n), (Input_1, \ldots, Input_j),$
$(InEvent_1, \ldots, InEvent_k), (Process_1, \ldots, Process_m), (Macro_1, \ldots, Macro_x),$
$(Mode_1, \ldots, Mode_w)) \quad \wedge$

$$(OutEvent'_1, \ldots, OutEvent'_y) = P((Attribute_1, \ldots, Attribute_n), (Input_1, \ldots, Input_j), (InEvent_1, \ldots, InEvent_k), (Process_1, \ldots, Process_m), (Macro_1, \ldots, Macro_x), (Mode_1, \ldots, Mode_w))$$

The postcondition of a mode indicates that the mode as a whole has changed state, including its subordinate Processes, Macros and Modes. Also changed as a result of the transformation process P are the Attribute, Input and Output values, and the InEvents and OutEvents of the mode. Input values and InEvents change as a result of the transformation process due to the possibility of feedback.

5.4 VHDL

5.4.1 VHDL Syntax. The object diagram for VHDL is shown in Figure 5.13. Since the object model for VHDL is relatively simple, we chose not to present portions of it at a time but in its entirety.

All items to be modelled in VHDL are classified as processes. A process can be further divided into entities or components. An entity is a stand-alone executable process within the VHDL environment. An entity is a complete application whose behavior can be modelled; it is not an architectural fragment of some larger construct. On the other hand, components can not stand alone or be executed by themselves; components are architectural fragments that are used in the development of larger constructs. We present some examples to emphasize the difference between VHDL entities and components:

- An *and* gate can be a entity where its behavior is simulated and modelled.
- A *J-K flipflop* can be an entity.

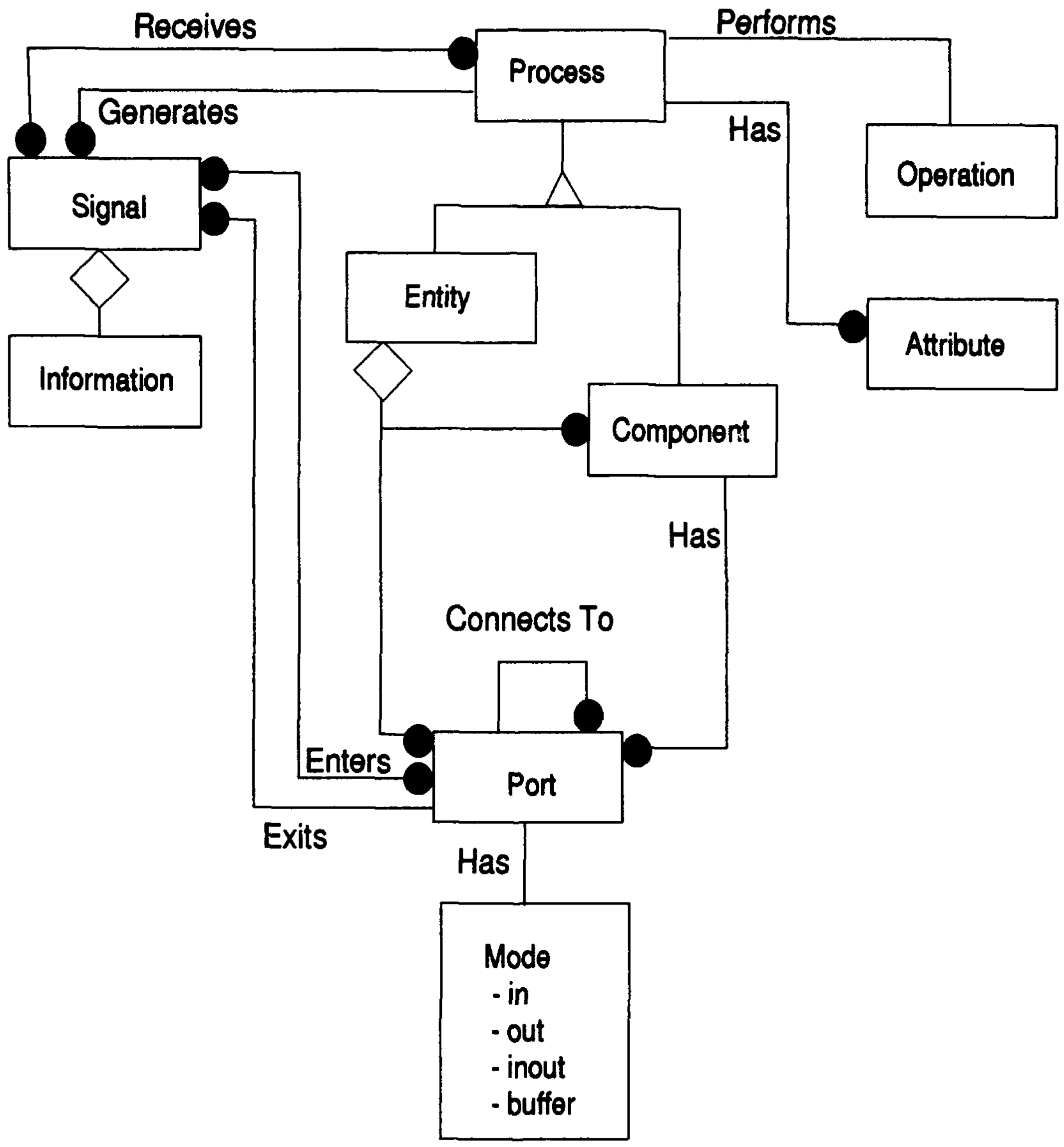


Figure 5.13 VHDL Object Model

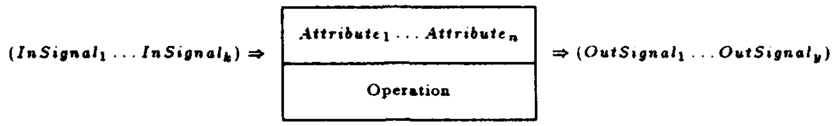

$(InSignal_1 \ldots InSignal_k) \Rightarrow$
$Attribute_1 \ldots Attribute_n$
Operation
$\Rightarrow (OutSignal_1 \ldots OutSignal_y)$

Figure 5.14 Behavior of a VHDL Component

- *Nand* and *nor* gates are considered components when composed together in the design of a *J-K flipflop*. The overall *J-K flipflop* circuit would still be considered an entity.
- If the *J-K flipflop* above is used in the development of an even larger application, it would then be a considered a component, and the larger application would be considered an entity.

The important difference between an entity and a component is level of abstraction. An entity is an application, while its identifiable subparts are components.

Information enters and exits a process via ports in VHDL. A port can have one of four modes: in, out, inout, and buffer. Processes share information by sending signals between processes. Signals are used to coordinate communication among all processes. Port connections are directional; that is, the information flows from one port to another in the direction specified by the port's mode (in, out, inout). Buffer, however is a special case of inout.

5.4.2 VHDL Semantics. Objects that exhibit interesting behavior within VHDL are the entity and component objects. These are the architectural entities whose semantics are analyzed.

5.4.2.1 Component. A component's semantics can be seen in Figure 5.14. Program P is the Operation identified in the box. Φ can be represented as the following:

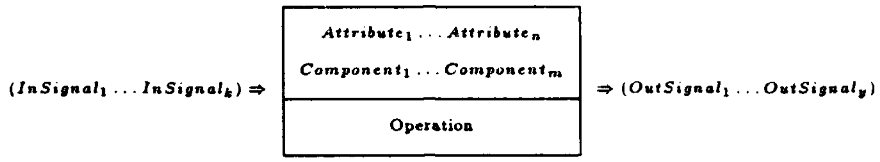

$(InSignal_1 \ldots InSignal_k) \Rightarrow$
$Attribute_1 \ldots Attribute_n$
$Component_1 \ldots Component_m$
Operation
$\Rightarrow (OutSignal_1 \ldots OutSignal_y)$

Figure 5.15 Behavior of a VHDL Entity

$$(ValidInput(InSignal_1) \wedge \ldots \wedge ValidInput(InSignal_k)) \wedge$$
$$\{Attribute_1, \ldots, Attribute_n\} \in ValidStateSpace$$

"ValidInput" for a VHDL component is a predicate that evaluates an InSignal to ensure that the InSignal is for this component to process. Also, "ValidStateSpace" is the cross product of all the legitimate state spaces that a component can occupy based on its Attributes. The precondition dictates that the InSignals are for this component and the values of its Attributes form a valid state space for the component.

As a result of the transformation process of P, Ψ is the following:

$$(Attribute'_1, \ldots, Attribute'_n) = P((Attribute_1, \ldots, Attribute_n),$$
$$(InSignal_1, \ldots, InSignal_k)) \quad \wedge$$
$$(InSignal'_1, \ldots, InSignal'_k) = P((Attribute_1, \ldots, Attribute_n),$$
$$(InSignal_1, \ldots, InSignal_k)) \quad \wedge$$
$$(OutSignal'_1, \ldots, OutSignal'_y) = P((Attribute_1, \ldots, Attribute_n),$$
$$(InSignal_1, \ldots, InSignal_k))$$

The postconditions imply that as a result of performing a transformation, the Attribute values, InSignals, and the OutSignals of a component have changed. Changes in the InSignals during this transformation are caused by the possibility of feedback.

5.4.2.2 Entity. The entity's semantics can be seen in Figure 5.15. Note that program P represents a distinct Operation (if the entity is not composed of subordinate Components) or the transformation of any of the entity's subordinate Components. Φ can be represented as the following:

$$(ValidInput(InSignal_1) \wedge \ldots \wedge ValidInput(InSignal_k)) \wedge$$
$$\{(Attribute_1, \ldots, Attribute_n), (Component_1, \ldots, Component_m)\} \in ValidStateSpace$$

The "ValidStateSpace" represents the cross product of all the legitimate state spaces that an entity can occupy based on its Attribute values and the state space of subordinate Components. The precondition above dictates that the InSignals are for this entity, and the state space defined by the Attributes and Components forms a valid state space for the entity.

After the transformation process of P, Ψ is:

$$\begin{aligned}
&(Attribute'_1, \ldots, Attribute'_n) = P((Attribute_1, \ldots, Attribute_n),\\
&\quad (InSignal_1, \ldots, InSignal_k), (Component_1, \ldots, Component_m)) \quad \wedge\\
&(Component'_1, \ldots, Component'_m) = P((Attribute_1, \ldots, Attribute_n),\\
&\quad (InSignal_1, \ldots, InSignal_k), (Component_1, \ldots, Component_m)) \quad \wedge\\
&(InSignal'_1, \ldots, InSignal'_k) = P((Attribute_1, \ldots, Attribute_n),\\
&\quad (InSignal_1, \ldots, InSignal_k), (Component_1, \ldots, Component_m)) \quad \wedge\\
&(OutSignal'_1, \ldots, OutSignal'_y) = P((Attribute_1, \ldots, Attribute_n),\\
&\quad (InSignal_1, \ldots, InSignal_k), (Component_1, \ldots, Component_m))
\end{aligned}$$

The postconditions imply that as a result of performing a transformation P, the Attribute values, subordinate Components, InSignals, and the OutSignals of an Entity have changed. The changing of the InSignals results from feedback.

5.5 μRapide

5.5.1 μRapide Syntax. There are four main features in μRapide: event patterns, components (and their corresponding interface), architectures, and mappings. Event patterns are used by other constructs to allow access to behavior, constraints, and mappings. An interface to a component defines how objects react to events, how objects change state, and how objects generate new events. Components can model a small object like a circuit gate or a large object like an airplane. Architectures on the other hand define the flow of events among components (objects). In other words, architectures define how components communicate by means of events. Finally, the similarity of one architecture to another is defined by a mapping; alternatively, mappings define how architectures are related. Figures 5.16 and 5.17 are object models for the component and the architecture, respectively. Figure 5.18 is a composite of all the architectural components of μRapide. As before, the dashed and solid lines in Figure 5.18 aid in discerning the different relationships in the object model.

5.5.2 μRapide Semantics. Objects that exhibit interesting behavior within μRapide are the component and architecture objects. These are the architectural items whose semantics are analyzed.

5.5.2.1 Component. A component's semantics can be seen in Figure 5.19. Note that program P represents the execution of a Method identified in the box. Φ can be represented as the following:

$$(ValidEvent(InEvent_1) \wedge \ldots \wedge ValidEvent(InEvent_k)) \wedge$$
$$\{(Attribute_1, \ldots, Attribute_n), (Constraint_1, \ldots, Constraint_m)\} \in ValidStateSpace$$

"ValidEvent" is a predicate that ensures that an InEvent is for this particular component. Also, "ValidStateSpace" is the cross product of all the legitimate state spaces that a component can occupy based on the values of its Attributes, as restricted by the Constraints. The above precondition implies that the InEvents received are for the component and the state space for a component is defined by its Attribute values and Constraints.

As a result of the transformation process of P, Ψ is the following:

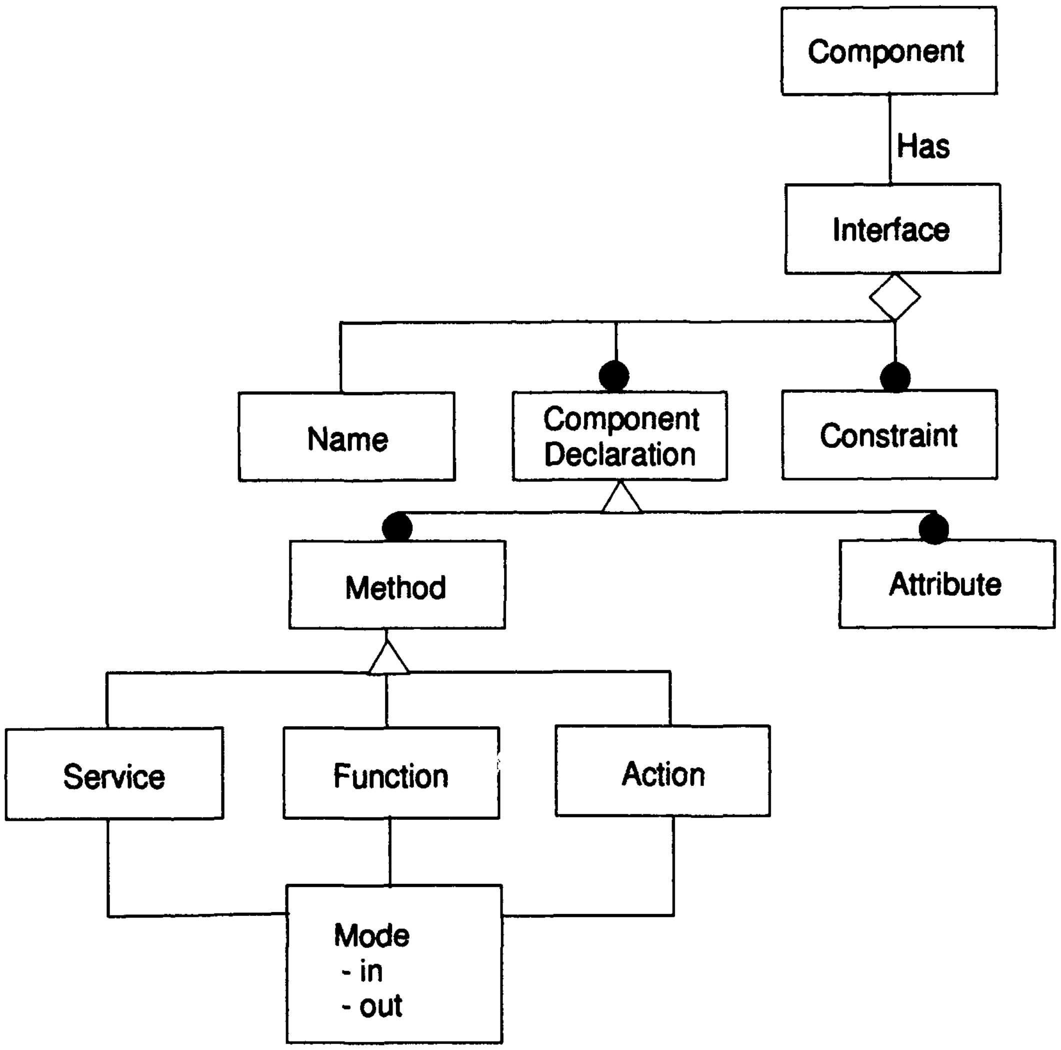


Figure 5.16 μRapide Component Object Model

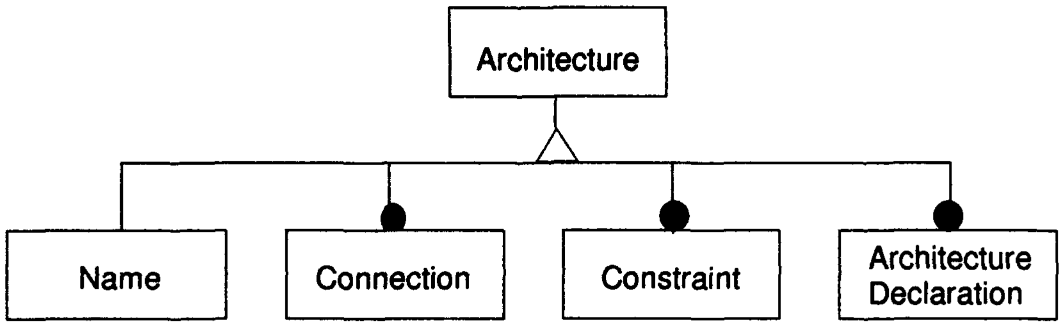


Figure 5.17 μRapide Architecture Object Model

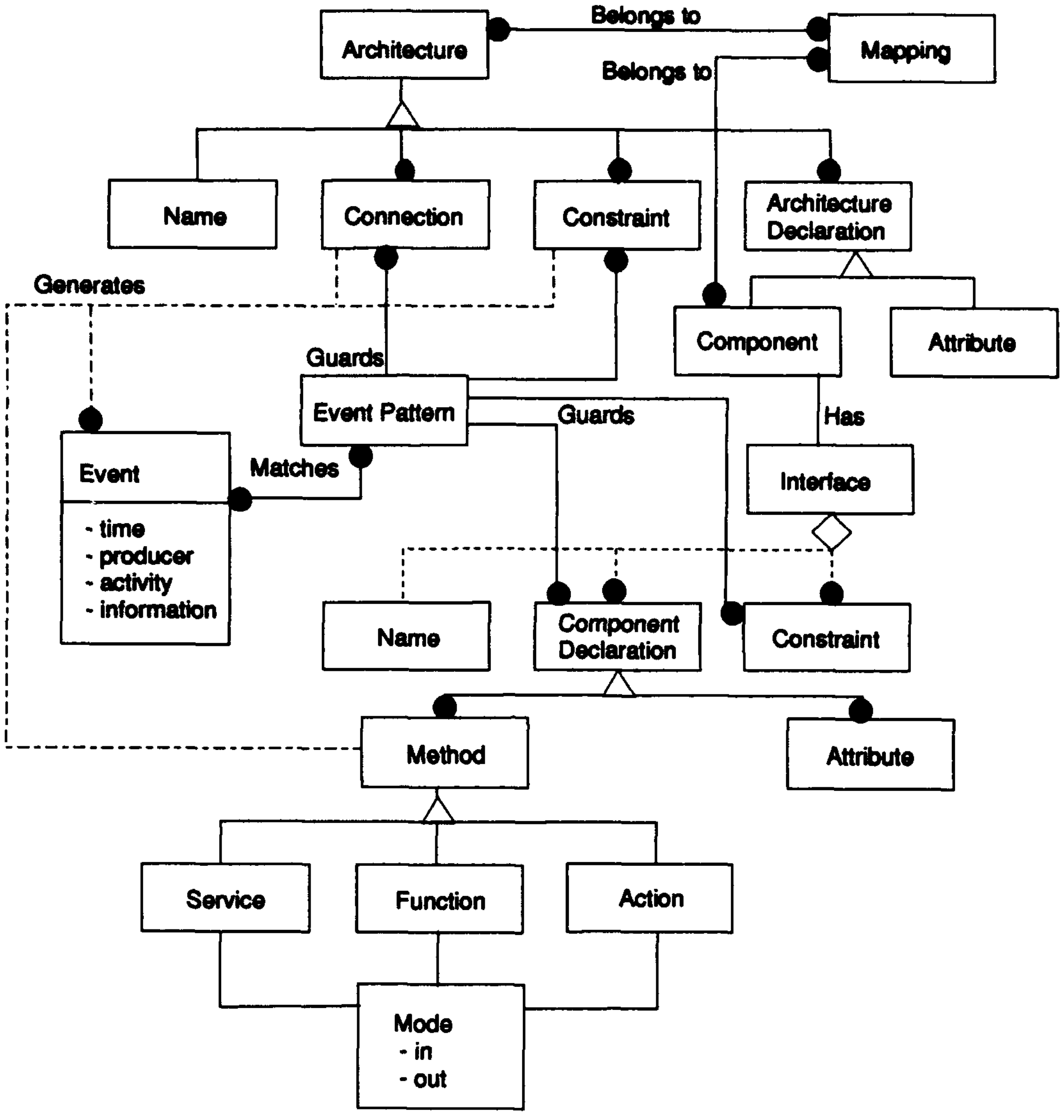


Figure 5.18 μRapide Complete Object Model

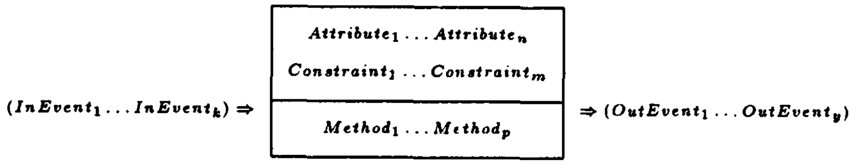

$Attribute_1 \ldots Attribute_n$
$Constraint_1 \ldots Constraint_m$
$(InEvent_1 \ldots InEvent_k) \Rightarrow$
$Method_1 \ldots Method_p$
$\Rightarrow (OutEvent_1 \ldots OutEvent_y)$

Figure 5.19 Behavior for a μRapide Component

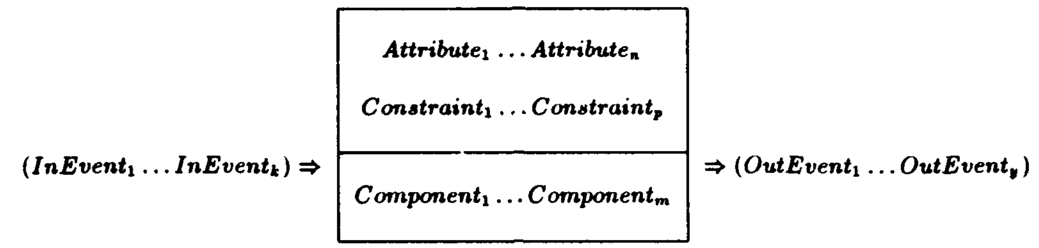

Attribute$_1$ ... Attribute$_n$
Constraint$_1$ ... Constraint$_p$
(InEvent$_1$ ... InEvent$_k$) ⇒
Component$_1$ ... Component$_m$
⇒ (OutEvent$_1$ ... OutEvent$_y$)

Figure 5.20 Behavior for a μRapide Architecture

$$(Attribute'_1, \ldots, Attribute'_n) = P((Attribute_1, \ldots, Attribute_n), (InEvent_1, \ldots, InEvent_k), (Constraint_1, \ldots, Constraint_m)) \quad \wedge$$
$$(InEvent'_1, \ldots, InEvent'_k) = P((Attribute_1, \ldots, Attribute_n), (InEvent_1, \ldots, InEvent_k), (Constraint_1, \ldots, Constraint_m)) \quad \wedge$$
$$(OutEvent'_1, \ldots, OutEvent'_y) = P((Attribute_1, \ldots, Attribute_n), (InEvent_1, \ldots, InEvent_k), (Constraint_1, \ldots, Constraint_m))$$

The postconditions imply that as a result of performing a transformation, the component may have changed state. The changes as a result of P are the Attribute values, the InEvents, and the OutEvents of a component.

5.5.2.2 Architecture. The architecture's semantics can be seen in Figure 5.20. Since an architecture is a higher-level abstraction built from lower level abstractions, its state space is defined by its local Attributes and Components. It should be noted that program P represents some subset of Methods of the subordinate components. Φ can be represented as the following:

$$(ValidEvent(InEvent_1) \wedge \ldots \wedge ValidEvent(InEvent_k)) \wedge \{(Attribute_1, \ldots, Attribute_n), (Component_1, \ldots, Component_m), (Constraint_1, \ldots, Constraint_m)\} \in ValidStateSpace$$

"ValidStateSpace" for an architecture is the cross product of all the legitimate state spaces that a architecture can occupy based on the values of its Attributes and its subordinate Components, as restricted by the Constraints. The above precondition requires that the InEvents received are for the architecture and the state space for an architecture is defined by its Attribute values, Components, and Constraints.

After the transformation process of P, Ψ is:

$$\begin{aligned}
&(Attribute'_1, \ldots, Attribute'_n) = P((Attribute_1, \ldots, Attribute_n),\\
&\quad (InEvent_1, \ldots, InEvent_k), (Constraint_1, \ldots, Constraint_m)) \quad \wedge\\
&(Component'_1, \ldots, Component'_m) = P((Attribute_1, \ldots, Attribute_n),\\
&\quad (InEvent_1, \ldots, InEvent_k), (Constraint_1, \ldots, Constraint_m)) \quad \wedge\\
&(InEvent'_1, \ldots, InEvent'_k) = P((Attribute_1, \ldots, Attribute_n),\\
&\quad (InEvent_1, \ldots, InEvent_k), (Constraint_1, \ldots, Constraint_m)) \quad \wedge\\
&(OutEvent'_1, \ldots, OutEvent'_y) = P((Attribute_1, \ldots, Attribute_n),\\
&\quad (InEvent_1, \ldots, InEvent_k), (Constraint_1, \ldots, Constraint_m))
\end{aligned}$$

The postconditions imply that as a result of performing a transformation, the architecture may have changed state. The changes as a result of P are the Attribute values, the subordinate Components, the InEvents, and the OutEvents of a Component.

5.6 IBM DSSA Using Batory's Layered Approach

5.6.1 Batory's Layered Approach Syntax. The object diagram for Batory's Layered Approach is shown in Figure 5.21.

The fundamental unit of software construction according to Batory is the component. Each component has an interface and an implementation. As part of Batory's architectural model, every component defined must belong to a realm. A realm contains a set of components with the same interface, but different implementations. In other words, every component within a realm inputs the same type of information, outputs the same type of information, but possibly performs a different transformation. A group of realms are used

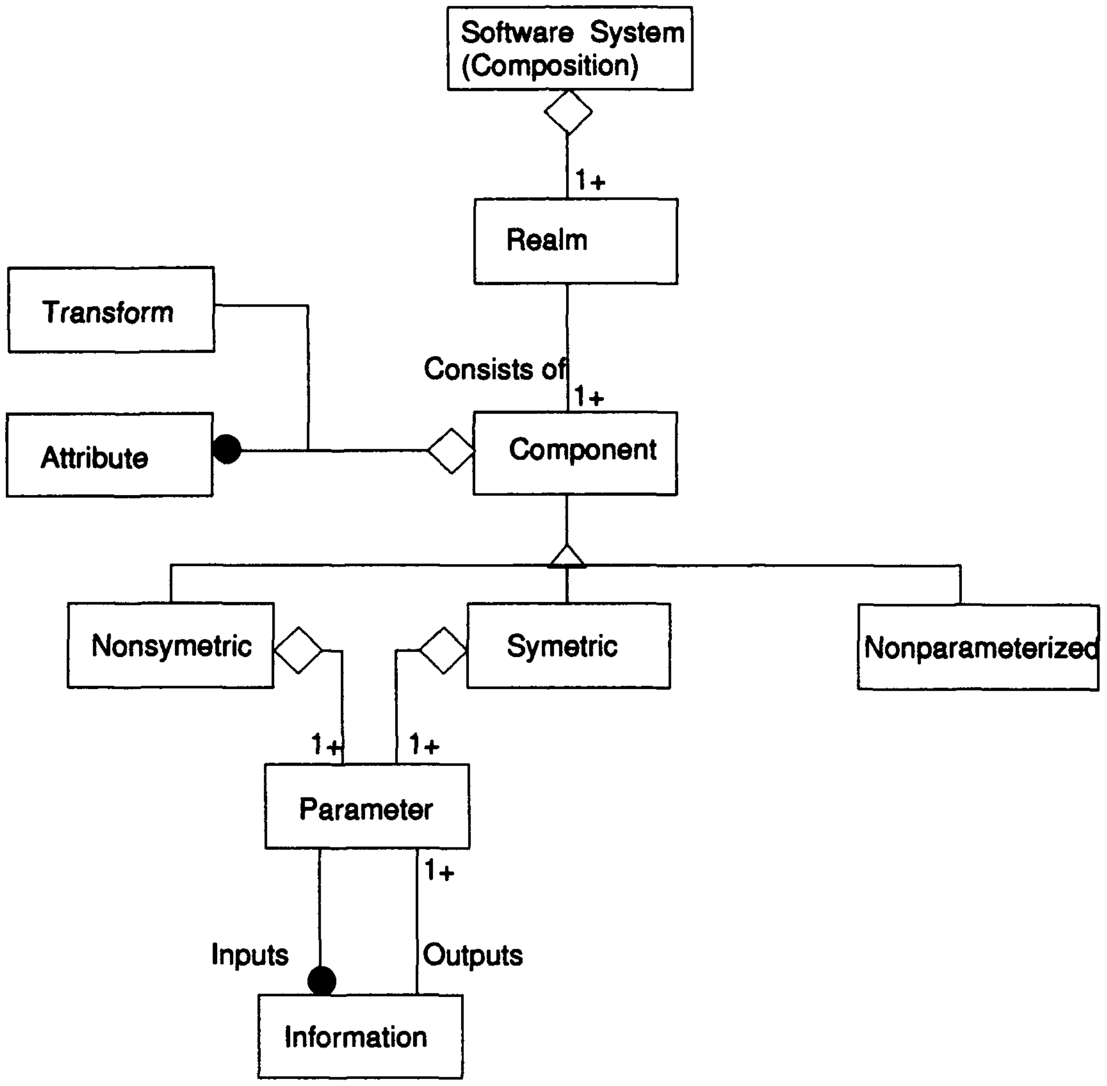


Figure 5.21 IBM DSSA Software Architectural Object Model

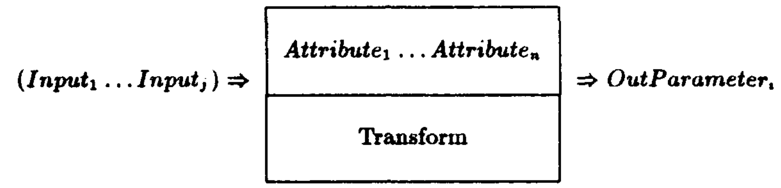
to define a composition, where a composition is a set of guidelines by which components are glued together to carry out a mission.

Components input information of a particular type from lower-level components and transform it to a type the next higher-level can use. A component can be symmetric or non-symmetric. Symmetric components have input parameters and their parameter input type is the same as their parameter output type. Having symmetric components within a software system allows for arbitrary stacking of components. On the other hand, non-symmetric components have input parameters, but their input parameter type and output parameter type do not match.

There are three ways of conceptually understanding components;

1. A component can be thought of as a layer, where a software system is a stacking of different layers (a composition of components).
2. A component can be thought of as a function that is invoked to provide some service. A software system is nothing more than a sequence of function calls, with the output of one function providing the input to the next function.
3. "The best analogy for realms and components is the concept of a grammar. A component corresponds to a production. Parameterized components are productions whose right-hand side reference nonterminals; parameterless components are productions that only reference terminals. Symmetric components correspond to recursive productions". (6)

5.6.2 Batory's Layered Approach Semantics. Objects that exhibit interesting behavior within Batory's Layered Approach are the components and software system (composition) objects. These are the architectural items whose semantics are analyzed.



$(Input_1 \ldots Input_j) \Rightarrow$
$Attribute_1 \ldots Attribute_n$
Transform
$\Rightarrow OutParameter_i$

Figure 5.22 Behavior of a IBM DSSA Parameterless Component

5.6.2.1 Parameterless Component. A parameterless component's semantics can be seen in Figure 5.22. It should be noted that program P is the Transform identified in the box. Φ can be represented as the following list:

$$(ValidInput(Input_1) \wedge \ldots \wedge ValidInput(Input_j)) \wedge$$
$$\{Attribute_1, \ldots, Attribute_n\} \in ValidStateSpace$$

"ValidInput" is a predicate that evaluates an Input and ensures that the input value being passed to the component is of a type the component can input and within bounds. Additionally, "ValidStateSpace" represents the cross product of all the legitimate state spaces that a component can occupy based on its Attribute values. The precondition above dictates that the Input values are of the correct type and range, and the values of its Attributes form a valid state space for the component.

As a result of the transformation process of P, Ψ will be the following:

$$(Attribute'_1, \ldots, Attribute'_n) = P((Attribute_1, \ldots, Attribute_n), (Input_1, \ldots, Input_j))$$
$$\wedge$$
$$OutParameter'_i = P((Attribute_1, \ldots, Attribute_n), (Input_1, \ldots, Input_j))$$

The postcondition of a component indicates that the component as a whole has changed state due to its change of Attribute values. Also changed as a result of the transformation process P is the OutParameter.

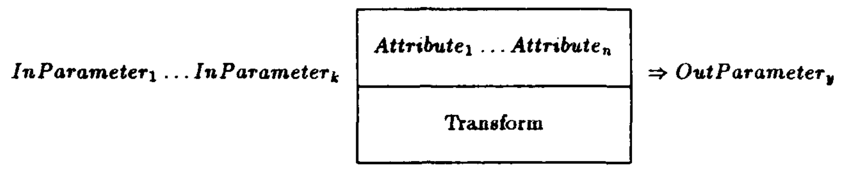

$InParameter_1 \ldots InParameter_k$
$Attribute_1 \ldots Attribute_n$
Transform
$\Rightarrow OutParameter_y$

Figure 5.23 Behavior of a IBM DSSA Parameterized Component

5.6.2.2 Parameterized Component. The parameterized component's semantics can be seen in Figure 5.23. It should be noted that program P is the Transform identified in the box. Φ can be represented as the following:

$$(ValidParameter(InParameter_1) \wedge \ldots \wedge ValidParameter(InParameter_k)) \wedge \\ \{Attribute_1, \ldots, Attribute_n\} \in ValidStateSpace$$

"ValidParameter" is a predicate that evaluates an InParameter and ensures that the InParameter is of a type the component can input. As with the parameterless component, "ValidStateSpace" represents the cross product of all the legitimate state spaces that a component can occupy based on its Attribute values. The precondition above requires that the InParameters are of the correct type, and the values of the Attributes form a valid state space for the component.

As a result of the transformation process of P, Ψ will be the following:

$$(Attribute'_1, \ldots, Attribute'_n) = P((Attribute_1, \ldots, Attribute_n), \\ (InParameter_1, \ldots, InParameter_k)) \quad \wedge \\ OutParameter'_i = P((Attribute_1, \ldots, Attribute_n), \\ (InParameter_1, \ldots, InParameter_k))$$

The postcondition of a component states that the component as a whole has changed state due to changes in the Attribute values. Also changed as a result of the transformation process P is the OutParameter.

$$(Input_1 \ldots Input_j) \Rightarrow \boxed{Component_1 \ldots Component_m} \Rightarrow (Output_1 \ldots Output_i)$$

Figure 5.24 Behavior of a IBM DSSA Software System

5.6.2.3 Software System. The software system's semantics can be seen in Figure 5.24. It should be noted that program P represents to transformation of any of its subordinate Components in the box. Φ can be represented as the following:

$$\begin{aligned}&(ValidInput(Input_1) \wedge \ldots \wedge ValidInput(Input_j)) \wedge \\ &\quad \{Component_1, \ldots, Component_m\} \in ValidStateSpace\end{aligned}$$

"ValidInput" is a predicate that evaluates an Input and ensures that the Input is of a type and range the software system can input. "ValidStateSpace" represents the cross product of all the legitimate state spaces that the software system can obtain based on the state spaces of its subordinate Components. The precondition above dictates that the Input Values are of the correct type and range, and the state space of its Components form a valid state space for the software system.

As a result of the transformation process of P, Ψ will be the following:

$$\begin{aligned}&(Component'_1, \ldots, Component'_m) = P((Component_1, \ldots, Component_m), \\ &\quad (Input_1, \ldots, Input_j)) \quad \wedge \\ &(Output'_1, \ldots, Output'_i) = P((Component_1, \ldots, Component_m), \\ &\quad (Input_1, \ldots, Input_j))\end{aligned}$$

The postcondition of a software system requires that the software system as a has changed state due to state changes of its Components. Also changed as a result of the transformation process P is the Output values.

5.7 Analysis of Software Architectures

Each section of this chapter analyzed a different software architecture using an object diagram and the axiomatic approach. The object diagrams allow the comparison of the structures of the software architectures while the axiomatic approach allows the comparison of semantics derived from the state space of the objects, as constrained by preconditions and postconditions.

5.7.1 Structural Analysis. Table 5.1 is a composite of the structural components of the software architectures that can be identified from the object diagrams and the documentation. Characteristics of the architectures are listed on the left, while the different architectures are listed across the top of the table.

As mentioned, the object diagram has been tailored for the purposes of this thesis effort. As such, some of the information presented in Table 5.1 has been reintroduced from the literature.

The following can be drawn from analyzing the object diagrams of the software architectures:

1. All architectural components have some way to retain their state information. In the lowest-level constructs, the state information is retained in attributes; in higher-level constructs, state information is captured in the subordinate objects, as well as attributes.

Table 5.1 Structural Commonalities of the Software Architectures

	OCU	MetaH	VHDL	μRapide	IBM DSSA
Validation Environment	Application Executive	Application	Test Bench	Rapide-1	Rapid Prototype
Events	Yes	Yes	Yes	Yes	Yes[1]
Event Management	Yes	Yes	Yes	Yes	Yes[1]
Event Buffers	Yes		Yes	Yes	Yes[1]
Lowest Abstraction	Primitive	Source Entity	Entity/ Component	Component	Parameterless Component
Higher Abstractions	Subsystem	Process, Macro, Mode	Entity	Component	Parameterized Component, Configuration, Software System
Attributes At Lowest Level	Yes	Yes	Yes	Yes	Yes
Attributes At Higher Level		Yes	Yes	Yes	Yes
Ways to Change State	Update Algorithm, Operations	Source Modules, Paths	Operations	Methods	Transforms
Data Ports	Yes	Yes	Yes		Yes[1]
State	Implicit	Implicit	Implicit	Implicit	Implicit

[1] Mentions, but gives no details.

2. All components of the different architectures have a way to change state; none are static. Some architectures use operations to change state, some use an update algorithm or an execution path, while others employ a transform.

3. All of the architectures incorporate events and event processing.

4. All constructs have a way to interface with their environment. Some components get their information from inputs, while others get their information from events.

5. All the software architectures analyzed employ a layered architecture. The layered architecture results in different levels of abstractions, with the higher-level abstractions requesting services from lower-level abstractions.

5.7.2 Semantics Analysis. Tables 5.2 and 5.3 are composites of the semantics for the most primitive objects (of interest) and the higher-level objects of interest within the architectures, respectively. By looking at each of the two categories of object types (most primitive and those "built on" the primitive types), we can see that there are differences within each category. As an example, all the primitive type objects change the state of their associating attributes except MetaH; a MetaH Source Entity does not change any of its local attributes. The same is true about the higher-level objects of the architectures; all change the state of their local attributes (if present) except MetaH.

The following are some general conclusions that may be drawn from the semantics (preconditions and postconditions) of the architectural components regardless, of the specific software architecture:

Table 5.2 Primitive Objects and Their Semantics

	OCU	MetaH	VHDL	μRapide	IBM DSSA
Objects That Change State					
Itself	Yes	Yes	Yes	Yes	Yes
Attributes	Yes		Yes	Yes	Yes
Output	Yes	Yes			Yes
Out Events	Yes	Yes	Yes	Yes	Yes[1]
Input	Yes	Yes			
In Events			Yes	Yes	Yes[1]

[1] Mentions, but gives no details.

Table 5.3 Higher Level Abstractions and Their Semantics

	OCU	MetaH	VHDL	μRapide	IBM DSSA
Objects That Change State					
Itself	Yes	Yes	Yes	Yes	Yes
Local Attributes			Yes	Yes	Yes
Subordinate Objects	Yes	Yes	Yes	Yes	
Output	Yes	Yes			Yes
Out Events	Yes	Yes	Yes	Yes	Yes[1]
Input	Yes	Yes			
In Events		Yes	Yes	Yes	Yes[1]

[1] Mentions, but gives no details.

1. The lowest-level constructs modify only their local attributes and may cause some type of outputs as a side effect of the transformation process.
2. The primitive constructs for the most part are passive; they initiate no actions on their own but service the requests of higher-level constructs.
3. The higher-level constructs not only change the state of their local attributes (if present) and possibly cause outputs, but may also force the change of state of their subordinate objects.
4. The higher-level construct's state is a composite of its local state and the state of its subordinates.
5. An object at a particular level is only able to modify the state of objects subordinate to it; it is not able to cause state changes to other objects at the same level or above in the architecture.

5.8 Conclusions From Analysis

As mentioned in Section 5.7.1, all of the architectures incorporate the layered approach in their design. An advantage of employing the layered approach is that it allows the developer of a system to abstract a component to the desired level. The developer can model an object at the lowest level or model the object as several subcomponents interacting to complete a task as a whole. An example of designing different levels of abstractions for the same object can be seen in VHDL and OCU: a flip-flop circuit can be modelled as a single object or may be built from lower-level objects such as *nor* gates or *nand* gates.

A common feature of all of these architectural models is the encapsulation of both the local state and information. This is accomplished via the layered approach and the semantics of the architectural model. The layered approach allows for the localization of the information, while the semantics of each model localizes the span of control. The localization is evident in the semantics of the architectural components of each architecture; each component changes its local state and may only change the state of objects subordinate to it.

There is no one-to-one mapping of components between software architectures. Although the components of two architectures may appear structurally and semantically similar, they cannot be directly substituted for each other; an example is an OCU primitive and a MetaH source entity. Both have ports to receive data from and send data through, both have attributes, and both are almost semantically identical. However, the substitution of a MetaH source entity for a primitive in the OCU structure would fail. The failure can be attributed to how each of these components change state; an OCU primitive changes state by executing one of its operations, where a MetaH source entity changes state by executing one of its source modules.

Another point that can be drawn from the analysis of these software architectural models is that the domain knowledge of the application area is captured in the most primitive objects of the architectures. The higher-level abstractions within an architecture have less domain knowledge of the application area; the information they do contain tends to identify their span of control, which subordinate objects are connected together, what resources are shared, and how they are to accomplish their mission.

By stepping back and looking over the information presented in Tables 5.1, 5.2, and 5.3, the architectures, their components, and their component's semantics all appear similar. From a structural standpoint, all the architectural components have ports or buffers to receive information, attributes to store information, lowest-level primitive abstractions, and a way to change state. From a semantic view, all the architectural components behave the same; they consume input or in events, they change state, and they produce new output or out events. One may conclude that there possibly exists a meta-structure (or meta-model) for software architectures.

Although the software architecture of all the architectural models studied in this thesis employ the layered approach, there are some conclusions that can be drawn from this effort that point to some necessary characteristics for all software architectures.

1. In order to localize the effects that changes have on a system and facilitate reuse within any architecture, the encapsulation of information (information hiding) and the limiting of the span of control of an architectural entity are good criteria for a software architecture. Employing these two concepts alone minimizes the effects of any changes on a system whether the system is being enhanced (adding new capabilities to a system) or maintained.

2. By limiting the span of control and encapsulation of the architectural entities, the preconditions and postconditions of the state space are limited as well. This restricting of the preconditions and postconditions of a construct makes it easier to replace an entity within one architecture with an entity from a different architecture that conceptually performs the same function but in a different way. This leads to the

possibility of reusing design and domain knowledge from other projects with different architectures.

Just as the characteristics presented above (limitation of the span of control and the encapsulation of information) are deemed good in an architecture, the lack of these characteristics may be deemed as bad. However, at present there does not exist a standard method to evaluate an architecture. Other areas of the software discipline have developed metrics to assess the code complexity (McCabe's Metrics), algorithms efficiency (O and σ), and test coverage (graph theory). It can be concluded that metrics for evaluating software architectures should be developed.

Finally, with a software architecture meta-model and a metric for evaluating software architectures, the capability exists to map a component of one architecture into the state space of another architecture. The metrics can be used to evaluate a software architecture from which an entity is to be drawn. The results of applying the metric can reveal the ease of removing the architectural component while allowing its behavior to remain intact. Likewise, the meta-model can be used to identify the components of the entity. After retrieving the entity, it can be mapped into the state space of the target architecture based on the preconditions and postconditions of the entity. The entity can be transformed without the loss of any knowledge by using a state-space transformation process as defined by (11). After the transformation process, the new entity is integrated into the target architecture.

VI. Detailed Design of Architect's Event-Driven Simulation Capability

6.1 Introduction

As presented in the operational concept (Section 3.4), events in Architect are passed to an independent subsystem by the executive. Each subordinate subsystem in the independent subsystem's structure interrogates the event after the event is received. The interrogation results in the event either being processed by the subsystem or being routed to another subordinate subsystem. The following sections describe the implementation of events within Architect and the subsequent modifications required to Architect's behavioral modeling capabilities. Additionally, these sections describe the information that is encapsulated in the events, purposes of the information within an event, and changes made to the subsystems to process and manage events.

6.1.1 Processing of Events. Before the structure of the events could be defined, event processing for Architect needed to be specified. For example, questions like would subsystems act as event managers with the primitives processing the events, or should the subsystems both process and manage events for its subordinate primitives? Keeping with the OCU model as defined by the SEI, subsystems are the "locus of the mission", while primitives are the "services to carry out the mission" (14:18). Since the subsystems are the locus of the mission, I opted to incorporate event processing into the controller of the subsystem. After the controller interprets an event, it invokes the appropriate subordinate object to carry out the service required. Resolving this issue allowed the identification of the target for an event (a primitive of a subsystem) and a routing scheme to the target.

6.1.2 Structure of Events. As mentioned in Chapter 3, there are two classes of events incorporated within the Architect system, application events and executive events. The following information was identified as being needed by all event types: the priority of the event, the event time, the target of the event and the path to the target (routing scheme) through the subsystem structures. Figure 6.1 shows the information common to all events. The ellipses imply that an event may contain additional information.

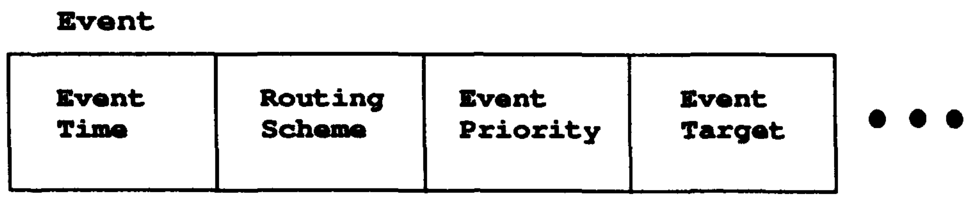


Figure 6.1 Common Information for All Events

Further, we needed to identify any requirements for domain-specific information that needed to be incorporated into any of the events. The only event identified needing domain-specific information was the SetState event. The SetState event contains the attribute names and values that need to be changed for a particular primitive at a future time. With no other information requirements for events, the application events were defined as:

1. Update event - received and interpreted by a subsystem. The target is a primitive of the subsystem that receives this event. The receiving subsystem invokes the Update function of the appropriate primitive.

2. SetState event - interpreted by a subsystem and used to invoke the SetState function of the appropriate primitive.

3. NewData event - received and interpreted only by the superior subsystem in an independent subsystem. The superior subsystem schedules Update events based on changes to any data in its import area. This event is discussed further in Section 6.1.5.2.

Figure 6.2 shows the events listed above and the information contained in each. The underlines emphasize differences between the NewData and Update event

Update Event

Event Time	Routing Scheme	Event Priority	Event Target
data type: Integer	data type: {subsystem}	data type: Integer	data type: Primitive

New Data Event

Event Time	Routing Scheme	Event Priority	Event Target
data type: Integer	data type: {subsystem}	data type: Integer	data type: Subsystem

Set State Event

Event Time	Routing Scheme	Event Priority	Event Target	Attribute values
data type: Integer	data type: {subsystem}	data type: Integer	data type: Primitive	data type: {Name-Value}

Figure 6.2 Application Events With Data Types

6.1.3 Event Driven Simulation Within Architect. The operational concept for events presented in Chapter III has the application executive passing events to subsystems. This is considerably different from the sequential flow of control that was originally designed in (3, 20). Subsystems now have to interpret and manage events in the event-driven simulation instead of performing a set of operations defined in an update algorithm. The flow of control for the event-driven mode is still sequential in nature; however, events are produced, consumed, routed, received and managed.

6.1.3.1 InEvent and OutEvent Areas. Data flow within the OCU model is managed through import and export areas. Since the SEI makes no reference to events or event processing within the OCU model, a means to pass events was required. Some of the architectures examined under this thesis separated event flow from data flow (25). Furthermore, in keeping with the OCU concept of arbitrary compositions of subsystems and primitives to form meaningful object abstractions, a decision was made not to incorporate events into the import and export areas. Event passing is implemented in much the same way that data passing is implemented in the OCU model. Events flow into a subsystem via an InEvent area and exit the subsystem via an OutEvent area. These additional constructs enhanced the state information of a subsystem by allowing the events of a subsystem to be persistent. Since event information is ultimately managed by the application executive's event manager (that is, a subsystem does not manipulate an event unless the target of the event is itself or a subordinate primitive), the InEvent and OutEvent areas are sets of events with no restrictions on the number or ordering of events within these areas. Under this effort, the InEvent area is implemented as a sequence of events; however, since there is just a single thread of control within an independent subsystem, the InEvent area will only contain one event. With the implementation of a truly concurrent simulation mode, more than one event may be present in the InEvent area. However, that is left for future research. An example of a subsystem with InEvent and OutEvent areas is shown in Figure 6.3.

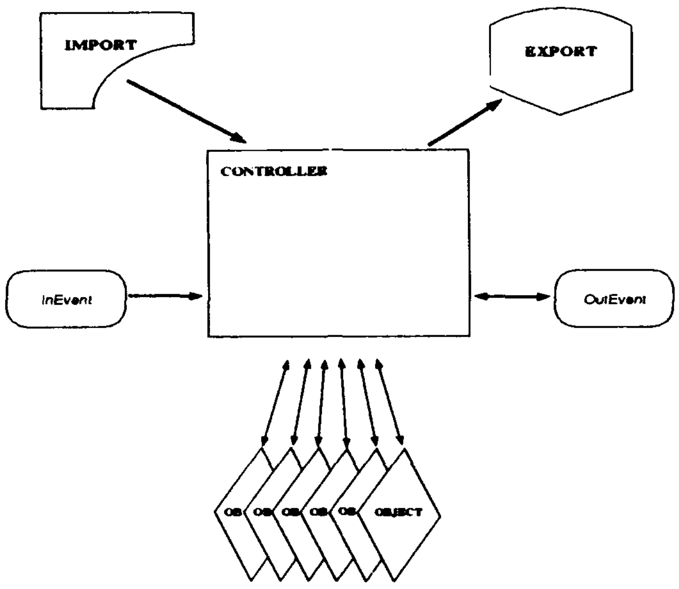


Figure 6.3 Subsystem With InEvent and OutEvent Areas

6.1.3.2 Flow of Events Through Subsystems. Section 6.1.1 indicated that the target of an event is usually a primitive object. The parent subsystem of the target primitive actually "decodes" the event and invokes the appropriate primitive's function as defined by the event type. The routing scheme of the event contains a sequence of subsystems that are superior to the primitive. Figure 6.4 shows a subsystem structure and a sample event targeted for a subordinate primitive; emphasized are the target and routing scheme fields of the event. The routing scheme within an event can be thought of as a directory to the primitive. Since the overall structure of the OCU model is hierarchical and the subsystems are only aware of subordinate objects directly beneath it, a subsystem must be able to query the event as to whether it needs to process or route the event. This querying is accomplished by the subsystem removing its name from the routing scheme. If there are any subsystems remaining in the routing scheme, then a subsystem routes this event to a subordinate subsystem, as specified by the next name in the routing scheme.

However, if the routing scheme is empty, the subsystem consumes the event and invokes one of its primitive's operations.

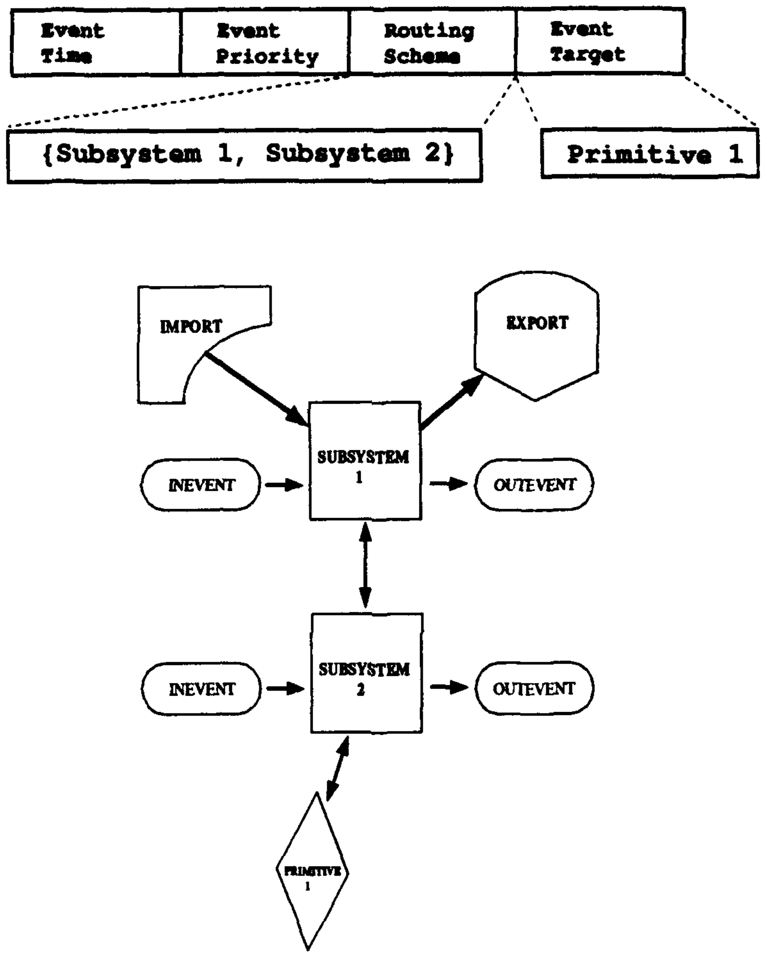


Figure 6.4 Sample Subsystem with Associated Event Targeted For Primitive

6.1.3.3 Changes to Execution Functions of Architect. This processing and managing of events forced Architect to have two EXECUTE functions. One EXECUTE function is the original simulation capability as implemented by (3, 20). The new EXECUTE function had to know of the InEvent and OutEvent areas of subsystems and the structure of the events. Additionally, the controller of a subsystem is no longer concerned with an Update Algorithm as the events received dictates the actions of the subsystem. The new execute function had to be capable of passing events down to subsystems and receiving events from lower level subsystems and primitives. This required a rewrite of the following Architect functions:

1. Execute-Subsystem - This function checks the subsystem's execution mode and either invokes the sequential mode as implemented by (3, 20) or invokes the Event-Driven-Controller.
2. Set-Export - This function checks the execution mode and either operates as it did under the sequential mode or returns a set of events.
3. Update - This function invokes the primitive objects update algorithm as it did under the sequential mode, but as a result of an event-driven execution, it returns a SetState event. This is determined by the domain implemented by the domain engineer.
4. SetState - Again, depending on the mode, this function works as it always did or returns a set of events.

Not only did some of Architect's original functions need to be rewritten, but some new functions had to be added to generate events:

1. Event-Driven-Controller - queries events of a subsystem to determine if this subsystem should process the event or route it to a subordinate subsystem.
2. Gen-Set-State-Event - generates a SetState event for a primitive.
3. Gen-Update-Event - generates an Update event for a primitive.

4. Gen-Transmit-Event - generates a Transmit event for the executive informing the executive some exports have changed.

5. Gen-Remove-Event - generates a Remove event for the executive informing the executive that a primitive wants to discard a previously generated event.

6. Eliminate-Dupes - reduces event handling overhead by reviewing all events received and eliminating redundant events based on the target and time fields within the event.

All of these functions are invoked when the event-driven EXECUTE-SUBSYSTEM function is invoked and the simulation mode is event-driven.

6.1.4 Changes to the Semantic Checks. With the inclusion of the event-driven simulation, the semantic checks were changed. A majority of the semantic checks are still valid and required. However, depending on the simulation mode, checks that validate the Update algorithm within a controller of a subsystem may or may not be [illegible]equired. Under the event-driven simulation, the presence of an Update algorithm i [illegible] al. Conversely, under the sequential mode, the Update algorithm is mandatory.

Also, since multiple independent subsystems are possible, the semantic checks that validate the import and export connections for data communication needed modification. Imports and exports within an independent subsystem are connected via source objects and target objects. These source and target are converses that identify either the export source that produces the information for this import or the import target that consumes the information from this export. With the implementation of the application executive, a connection object is now valid between the imports and exports in addition to the source and target objects. A connection object serves the same purpose as the source and target objects; however, a connection object connects the imports and exports of independent subsystems. For a detailed description of the connection object see (29). Figure 6.5 shows the relationships of source, target and connection objects.

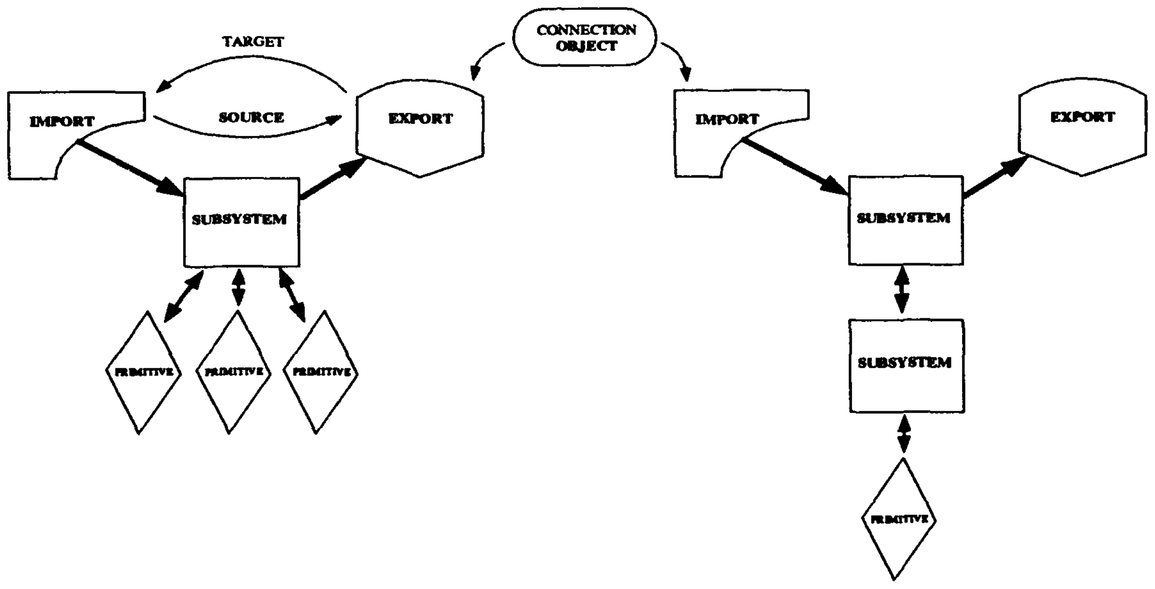


Figure 6.5 Two Independent Subsystems showing the Relationships of Source, Target and Connection Objects

6.1.5 Consolidation of Import Areas and Export Areas. As a result of the implementation of the application executive functions within Architect (29), Architect must be capable of handling multiple independent subsystems. Not only is it possible for the application specialist to generate subsystems for the application being designed, but he can also generate executive subsystems as well. Since the executive subsystems are essentially independent of the application subsystems, there does not exist a parent-child relationship between these two subsystems. An application subsystem and an executive subsystem are independent subsystems that communicate; they are essential to modeling the behavior of an application, but neither is subordinate to the other.

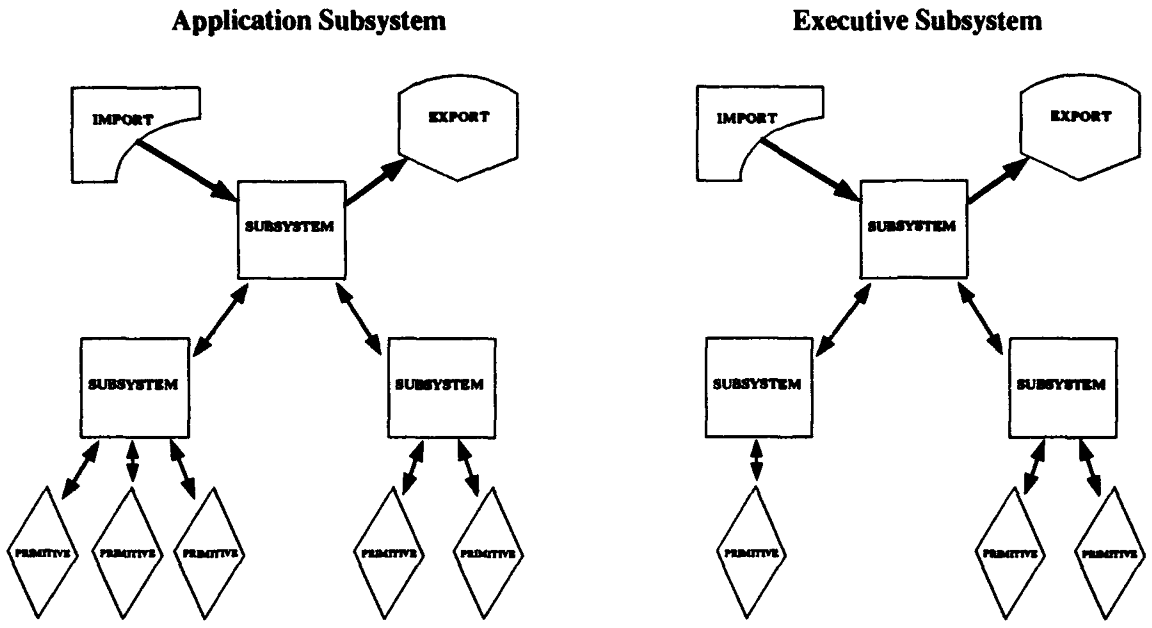


Figure 6.6 Independent Subsystems Represented as an Executive Subsystem and an Application Subsystem

6.1.5.1 Relocation of Import and Export Areas. These independent subsystems forced a change to the original implementation of the import areas and export areas. As originally implemented, import and export areas were associated with each subsystem at each level of a subsystem's hierarchy (see Figure 6.7). This worked fine when composing an application consisting of a single independent subsystem; however, with multiple independent subsystems, this implementation was no longer feasible. Allowing multiple independent subsystems to connect to subordinate subsystems of other independent subsystems obligated subsystems to know more about their environment than intended by the SEI (14:18). The OCU model was designed so that each level of abstraction within a subsystem only knew of its immediate subordinate objects. To resolve the contention, the import areas and export areas were consolidated at the highest-level subsystem in an independent subsystem, thus limiting the knowledge level of the lower-level subsystems and primitives. Figure 6.6 above shows how the imports and exports have been consolidated at the highest-level subsystem in an independent subsystem. This is strictly an implementation detail; conceptually, the imports and exports are the same as the SEI intended.

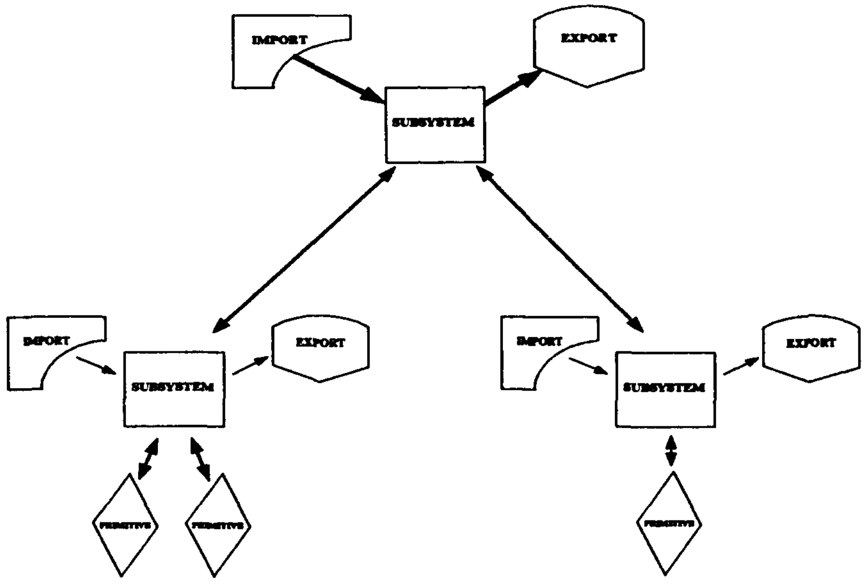


Figure 6.7 Original Import Export Implementation

6.1.5.2 New Information in the Imports and Exports. Since the import area and export area are at the highest-level subsystem, only the appropriate subsystem and primitive can be allowed to access its import or export data. This was accomplished by adding the owning subsystem information to the import and export areas and partitioning the import and export areas.

Further, a decision was made that the highest-level subsystem manages access to all the imports and exports of all of its subordinate primitives. This management was required since the executive could now change the data in an import area as a result of some type of communication from another independent subsystem. When the data of an import area is changed, the executive informs the highest-level subsystem of an independent subsystem of the new data via a NewData event. Reception of a NewData event signals the highest-level subsystem that some other independent subsystem has communicated with it, and it needs to schedule some events based on the new data in its import area. The highest-level subsystem only schedules update events; it collects the information needed to build the update events directly from the information present in its import area. In keeping with the SEI's intentions of anonymity of subsystems, the information required to generate an update event had to be moved to the import area. The knowledge of a subsystem is still limited, since a highest-level subsystem does not explicitly know of any subordinate objects more than one level beneath it. The information required is present in the import area; it is the routing scheme needed to help route an event to its intended target. Figures 6.8 shows a sample subsystem structure, and Figure 6.9 shows the information contained as part of the imports and exports of the subsystem, respectively.

6.2 Conclusion

This chapter presented an overview of the detailed design of incorporating event processing and management into Architect. Since Architect was designed around the OCU software architectural model, the inclusion of an event processing capability had to conform to the intent and spirit of the OCU model as defined by the SEI (14). All

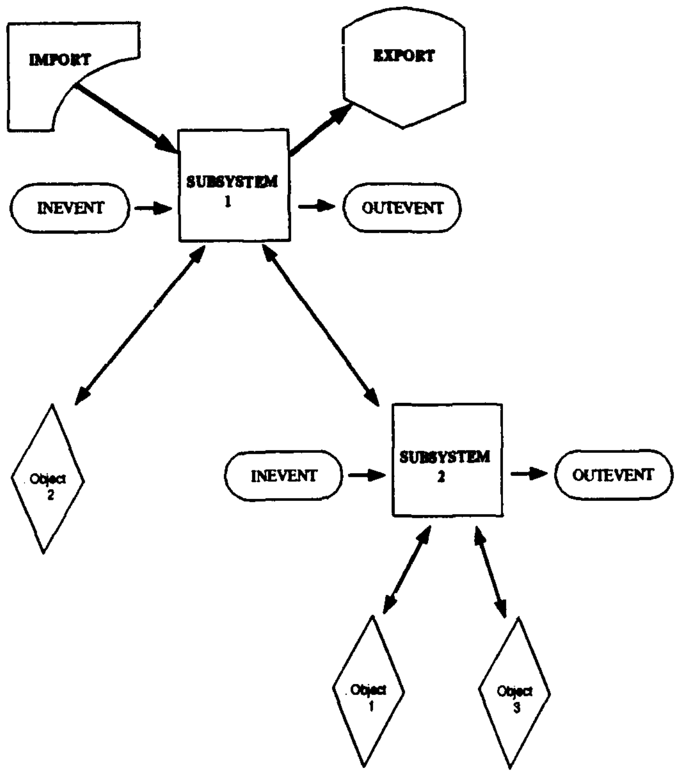
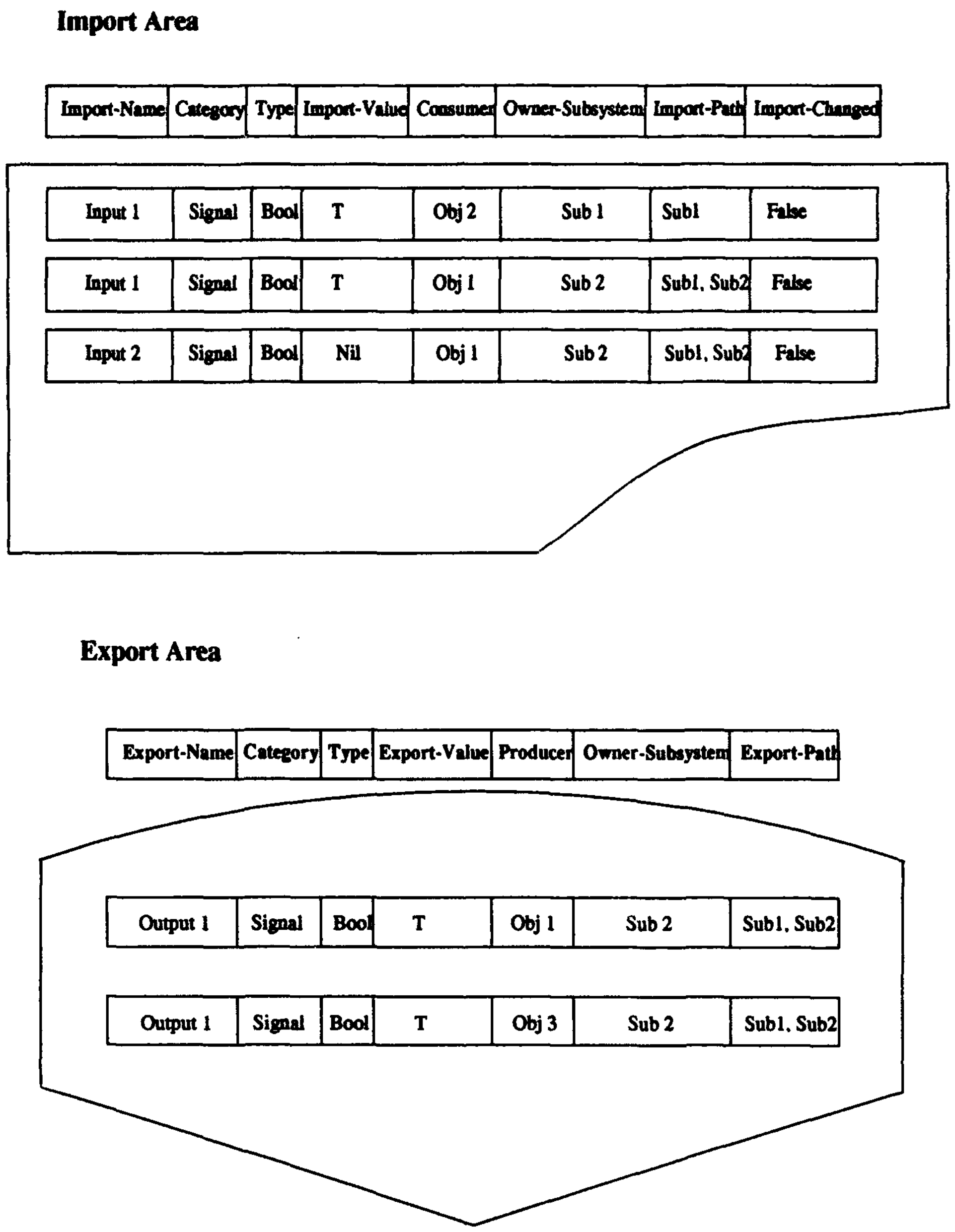


Figure 6.8 Sample Subsystem

decisions involving event processing have been presented in this chapter. The next chapter shows the applications used to validate the changes to the import and export areas and the applications used to verify the event-driven simulation capability. All the changes were validated using domains available as a result of other research efforts (26, 27).



Import Area

Import-Name	Category	Type	Import-Value	Consumer	Owner-Subsystem	Import-Path	Import-Changed
Input 1	Signal	Bool	T	Obj 2	Sub 1	Sub1	False
Input 1	Signal	Bool	T	Obj 1	Sub 2	Sub1, Sub2	False
Input 2	Signal	Bool	Nil	Obj 1	Sub 2	Sub1, Sub2	False

Export Area

Export-Name	Category	Type	Export-Value	Producer	Owner-Subsystem	Export-Path
Output 1	Signal	Bool	T	Obj 1	Sub 2	Sub1, Sub2
Output 1	Signal	Bool	T	Obj 3	Sub 2	Sub1, Sub2

Figure 6.9 Sample Import/Export Information and Data

VII. Validation of Architect's Event-Driven Capability

7.1 Introduction

The event-driven capability of Architect was validated in multiple phases. One phase of the validation was done using the primitives implemented as part of the original circuits domain (3, 20); the other phase of validation was done using the application executive implemented by (29) and the new domains incorporated by (26). A multi-phase approach was the result of the development time of the application executive and the development time of the new domains. Since the consolidation of imports and exports to the highest-level subsystem was accomplished independently of the implementation of the event-driven capability, these changes were also validated independently. As more of the event-driven capability was realized, incremental validation was performed to validate this new capability. The rest of this chapter discusses the results of the validation.

7.2 Consolidation of Import and Export Areas

The validation of the consolidated import and export areas was completed using the original gate primitives developed by (3, 20). Two scenarios for validation were performed, one using a single independent subsystem and the other using multiple independent subsystems.

7.2.1 Single Independent Subsystem. This validation scenario consisted of two different circuits. The first circuit was a simple design, as seen in Figure 7.1; Figure 7.2 shows some of the multiple configurations this circuit may have using subsystems and primitives. The truth-table for the simple circuit is shown in Table 7.1. This circuit helped

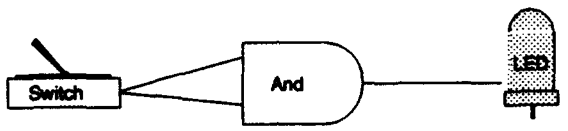


Figure 7.1 Simple Circuit for Independent Subsystem

Table 7.1 Truth-Table of Single Independent Circuit

Switch	LED
0	Off
1	On

by initially focusing on the complex interactions of a few subsystems and primitives, and their respective imports and exports vice the large volume of interactions in a more complex structure.

The next circuit used to test the import/export consolidation was more ambitious. The circuit was a binary array multiplier and is shown in Figure 7.3. The configuration of the subsystems and primitives for the binary array multiplier may be seen in Figure 7.4 and the truth table is in Table 7.2.

7.2.2 Multiple Independent Subsystems. Since Architect was soon to have multiple independent subsystems as part of an application, this validation included a circuit that passed information between two independent subsystems. The circuit chosen was the same circuit used in Figure 7.1, but its implementation was changed as illustrated in Figure 7.5.

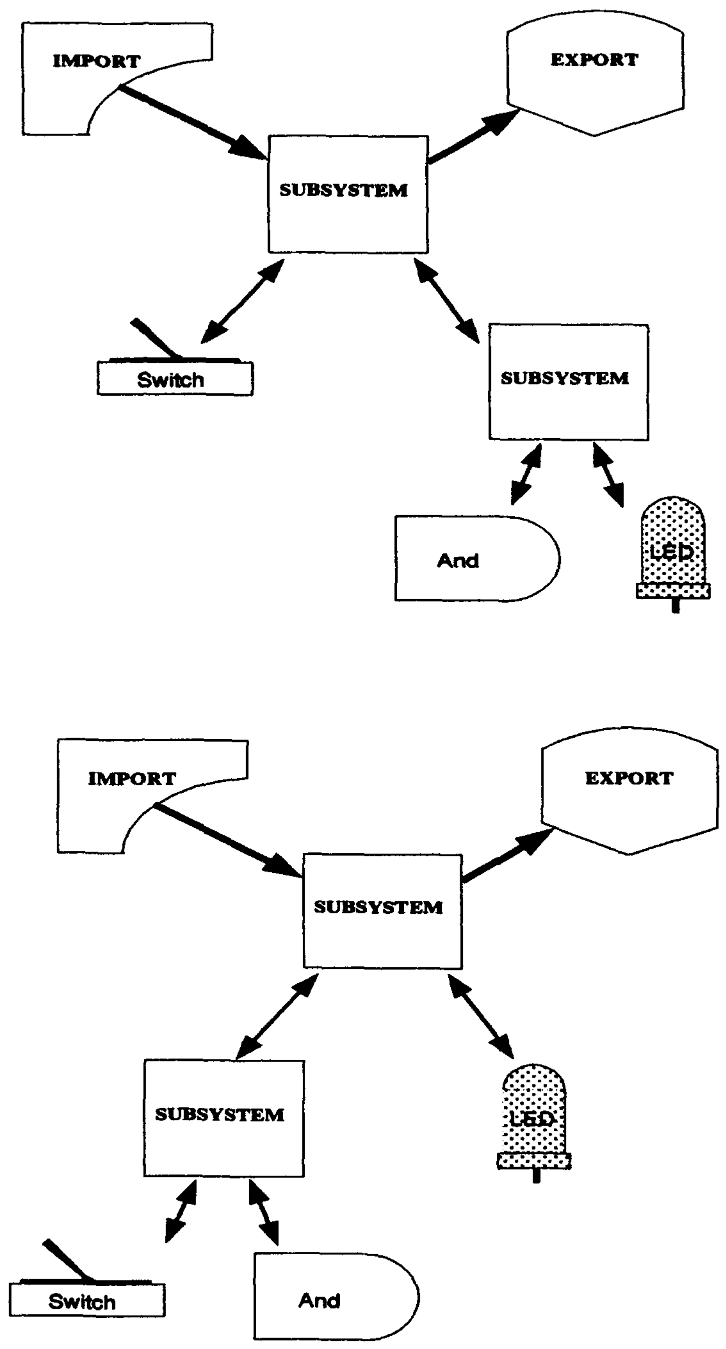


Figure 7.2 Simple Circuit as an Architect Implementation

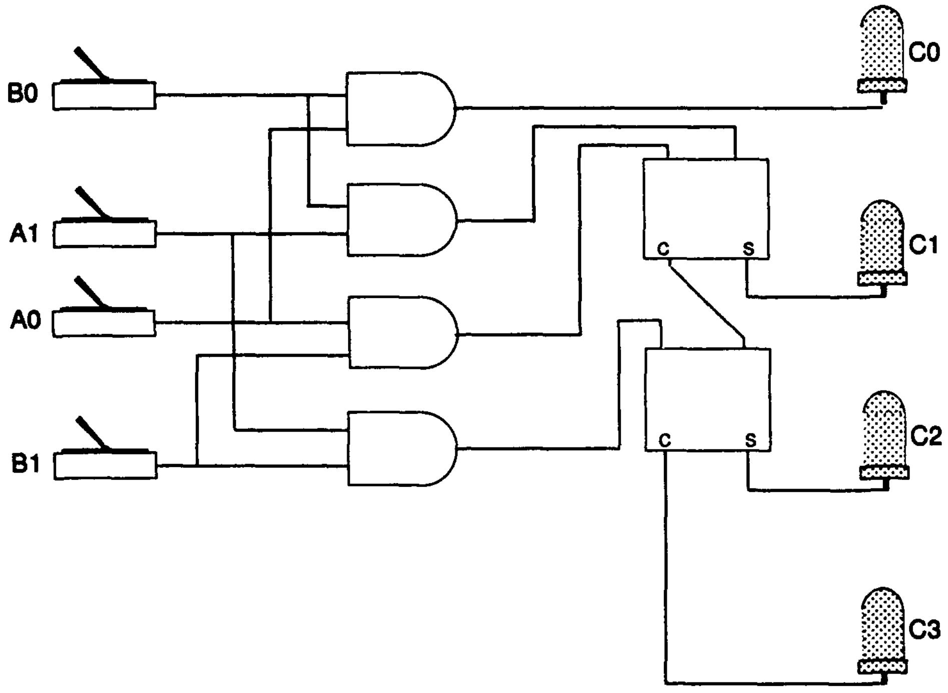


Figure 7.3 Binary Array Multiplier Circuit

Table 7.2 Truth-Table of the Binary Array Multiplier Circuit

B0	B1	A1	A0	C3	C2	C1	C0
0	0	0	0	Off	Off	Off	Off
0	0	0	1	Off	Off	Off	Off
0	0	1	0	Off	Off	Off	Off
0	0	1	1	Off	Off	Off	Off
0	1	0	0	Off	Off	Off	Off
0	1	0	1	Off	Off	Off	On
0	1	1	0	Off	Off	On	Off
0	1	1	1	Off	Off	On	On
1	0	0	0	Off	Off	Off	Off
1	0	0	1	Off	Off	On	Off
1	0	1	0	Off	On	Off	Off
1	0	1	1	Off	On	On	Off
1	1	0	0	Off	Off	Off	Off
1	1	0	1	Off	Off	On	On
1	1	1	0	Off	On	On	Off
1	1	1	1	On	Off	Off	On

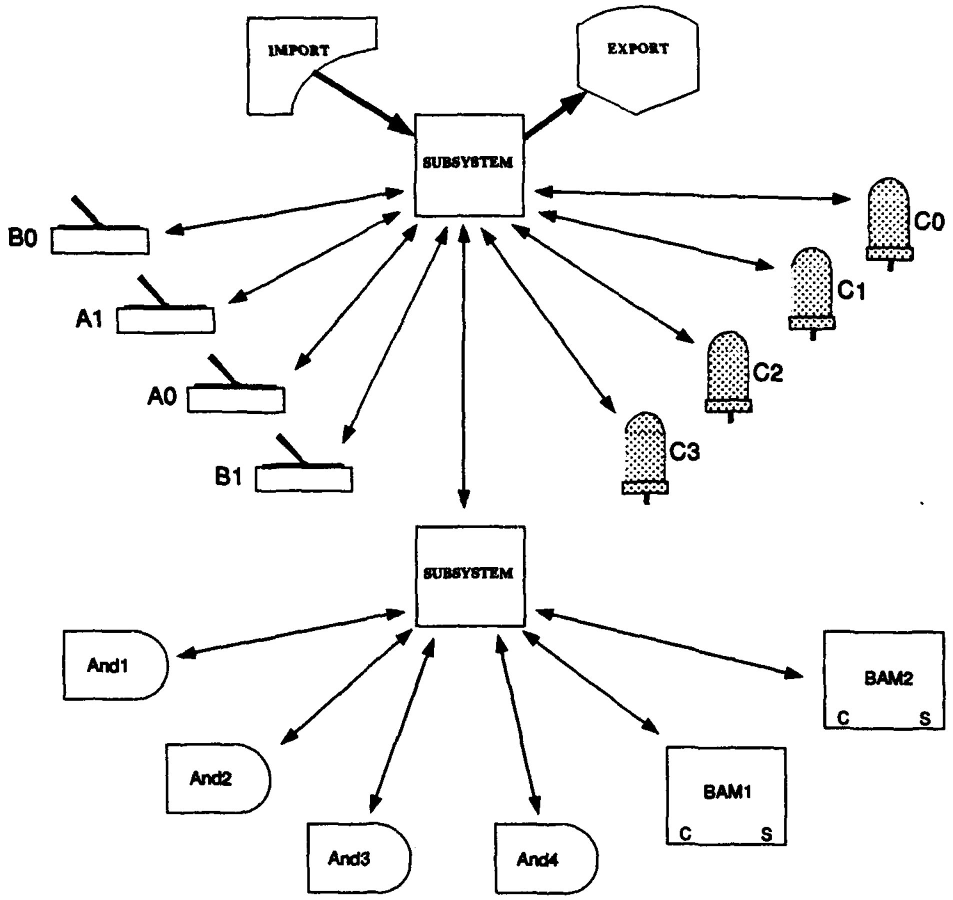


Figure 7.4 Architect Implementation of the Binary Array Multiplier

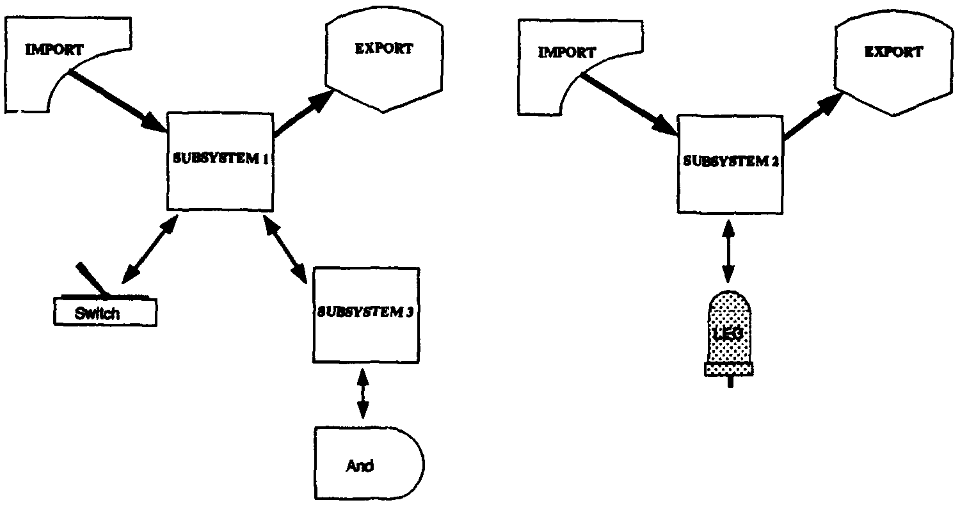


Figure 7.5 Architect Implementation of Simple Circuit with Independent Subsystems

7.3 Initial Validation of the Event-Driven Capability

Due to the timing of the development of the application executive and the new domain, the event-driven capability was initially tested using four primitives from the new domain and a prototype application executive stub. This prototype executive was nothing more than a pseudo application executive to provide an event manager capability to simulate the behavior of the application, since the application executive being developed as part of (29) was not ready for validation. The initial set of primitives provided by the circuits event-driven domain were a *switch, or gate, and gate*, and *LED.* With the initially limited capabilities of the validation software, the first circuits used for event-driven validation were rudimentary. However, this independent validation allowed for loose coupling between the forthcoming application executive and the event processing capabilities of subsystems.

7.3.1 Single Independent Subsystem. The single independent circuit to test the event-driven capability was again the simple circuit use in Figure 7.1. The proposed events that needed to be generated and consumed as a result of simulating the circuit are shown in Table 7.3. The events in the table are in the order that they are generated and consumed. Thus, an Update event for And is generated before the Transmit event is (a Transmit event is an executive event and is discussed in (29)). Likewise, the Update event is consumed before the Transmit event is. The data under the "Information for Executive" field is data required by the application executive but not used by the "mock-up" executive. The SetState event above the double line is obtained from the application specialist during the application definition process; it was not generated as part of the event-driven simulation.

Table 7.3 Events and Their Order for Single Independent Subsystem

Event Type	Target	Information for Executive
SetState	Switch	
Update	And	
Transmit		Out 1, Switch
SetState	And	
Update	LED	
Transmit		Out 1, And
SetState	LED	

Table 7.4 Events and Their Order for Multiple Independent Subsystems

Event Type	Target	Information for Executive
SetState	Switch	
Update	And	
Transmit		Out 1, Switch
SetState	And	
Transmit		Out 1, And
NewData	Subsystem2	
Update	LED	
SetState	LED	

7.3.2 Multiple Independent Subsystems. The multiple independent circuit to test the event-driven capability was the same circuit presented in Section 7.3.1 but implemented in Architect as Figure 7.5. The proposed events that needed to be generated and consumed as a result of simulating the circuit are shown in Table 7.4. As before, the SetState event above the double line is obtained from the application specialist during the application definition process; it was not generated as part of the event-driven simulation.

7.4 Final Validation of the Event-Driven Capability

7.4.1 Single Independent Subsystem. The single independent circuit to test the final event-driven capability is shown in Figure 7.6 and its corresponding Architect implementation in Figure 7.7. The truth-table is contained in Table 7.5, and the proposed

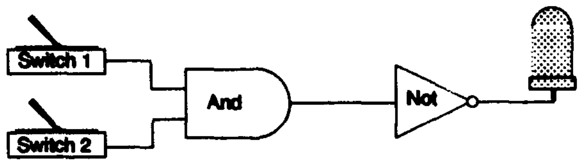


Figure 7.6 Simple Circuit for Event-Driven Validation

Table 7.5 Truth-Table of Simple Circuit for Initial Event-Driven Validation

Switch 2	Switch 1	Led
0	0	On
0	1	On
1	0	On
1	1	Off

events and their order for generation and consumption as a result of simulating the circuit are shown in Table 7.6. Once again, the SetState events above the double line are input as part of the application definition process, they were not generated as part of the event-driven simulation.

7.4.2 Multiple Independent Subsystems. The multiple independent circuit to test the final event-driven capability was not much more involved than the circuit in Section 7.4.1, and it is shown in Figure 7.8. However, the Architect implementation of the circuit was much more involved, as shown in Figure 7.9. The truth-table for the circuit in Figure 7.6 is contained in Table 7.7. The proposed events and their order of consumption as a result of simulating the circuit are contained in Table 7.8. The SetState events above the double line are obtained from the application specialist as part of the application definition. Since there are multiple independent subsystems in this simulation, the ordering of the generation of events may vary; however, what is important is the fact that the events were generated and the events were consumed in the proper order. The lines separating

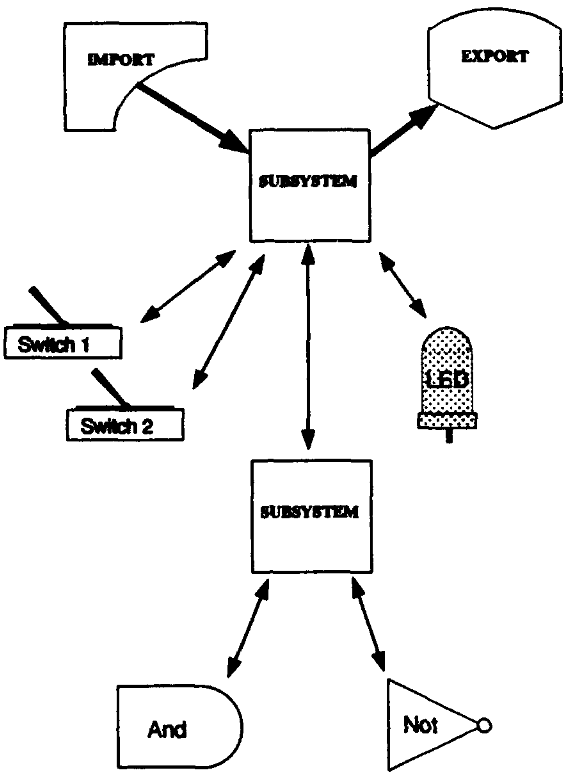


Figure 7.7 Simple Circuit for Event-Driven Validation in Architect

Table 7.6 Events and Their Order For Final Simple Circuit Validation

Event Type	Target	Information for Executive
SetState	Switch 1	
SetState	Switch 2	
Update	And	
Transmit		Out 1, Switch 1
Transmit		Out 1, Switch 2
SetState	And	
Update	Not	
Transmit		Out 1, And
SetState	Not	
Update	LED	
Transmit		Out 1, Not
SetState	LED	

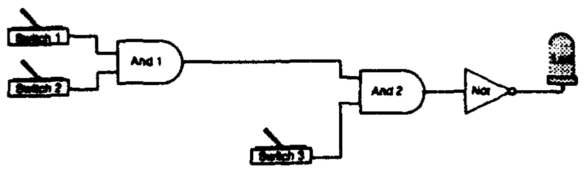


Figure 7.8 Enhanced Circuit for Event-Driven Validation

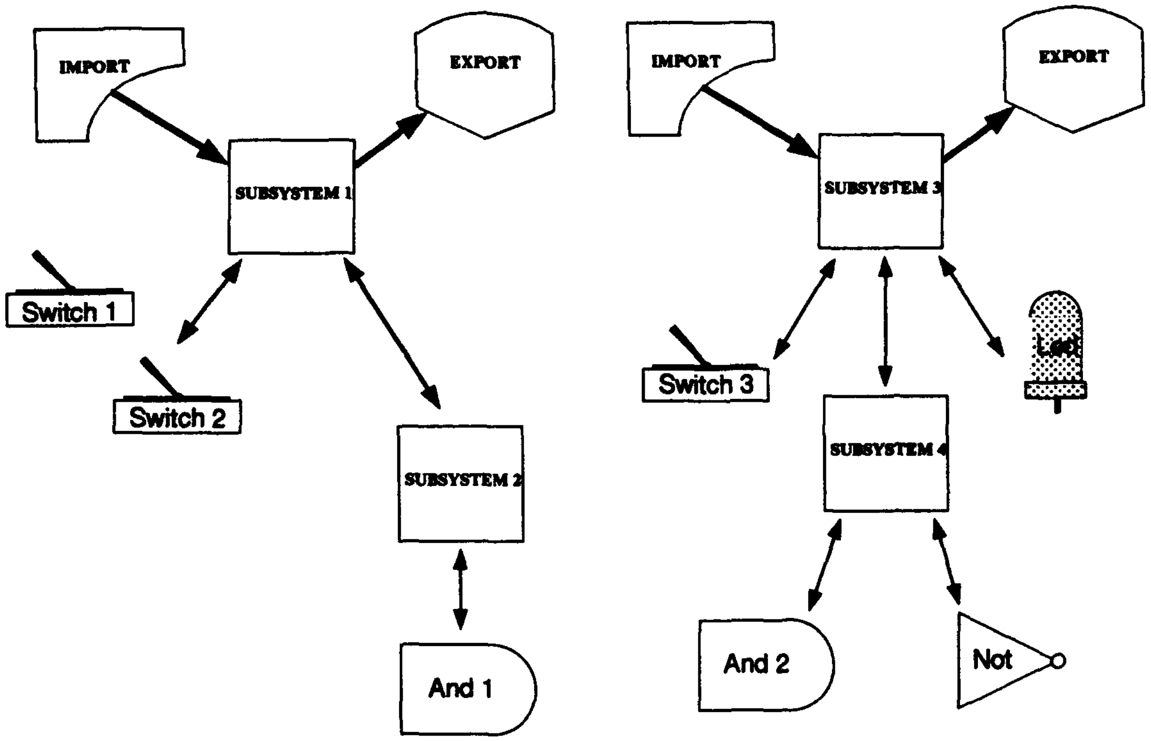


Figure 7.9 Enhanced Circuit for Event-Driven Validation in Architect

the events show what events occur together or at a single point in time. As an example, the first five events after the double line are executed at a single point in time during the simulation; the next three events occur at some future point of time in the simulation.

7.5 Conclusions

The multi-phase approach for validation and verification of Architect's new capabilities proved beneficial for several reasons:

Table 7.7 Truth-Table for Enhanced Circuit Validation

Switch 3	Switch 2	Switch 1	Led
0	0	0	On
0	0	1	On
0	1	0	On
0	1	1	On
1	0	0	On
1	0	1	On
1	1	0	On
1	1	1	Off

Table 7.8 Events and Their Order For Final Enhanced Circuit Validation

Event Type	Target	Information for Executive
SetState	Switch 1	
SetState	Switch 2	
SetState	Switch 3	
Update	And 1	
Transmit		Out 1, Switch 1
Transmit		Out 1, Switch 2
Update	And 2	
Transmit		Out 1, Switch 3
SetState	And 1	
Transmit		Out 1, And 1
SetState	And 2	
NewData	Subsystem 3	
Update	And 2	
SetState	And 2	
Update	Not	
Transmit		Out 1, And 2
SetState	Not	
Update	LED	
Transmit		Out 1, Not
SetState	LED	

1. The initial phase allowed for the overlap of different development efforts for the new domains, application executive, and event-driven simulation capabilities (29, 26).

2. The initial phase of moving the import/export areas was completed prior to any of the other efforts; as such, the code was used as the baseline for the application executive and the development of new domain primitives.

3. The application "mock-up" and new domain primitives resulted in the event-driven capabilities for Architect being available prior to full implementation of the application executive. Again, this code was used as a baseline for further development.

4. The small increments of changes resulted in focusing on the task at hand. Errors could be isolated and repaired quickly by being able to identify what code had introduced the error.

5. The small increments also resulted in a staged development. As each phase of development was validated, the next phase was developed using previously developed tools or functions.

6. Lastly, during final integration of the event-driven capability, application executive and new domains, errors were quickly isolated.

VIII. Conclusions and Recommendations

8.1 Summary of This Research

The primary purpose of this research was to develop a framework for comparing software architectures. The framework had to be capable of capturing the salient points of a software architecture and being applied to several architectures without tailoring.

A secondary effort was to modify the simulation capability of a prototype domain-oriented application composition system called Architect, to include events, event handling, and event management.

8.1.1 Framework for Comparing Software Architectures.

The primary purpose for developing a framework for comparing software architectures was to evaluate the OCU model for its suitability as a software architectural model for application composition and generation systems. A secondary purpose was to identify potential sources for design reuse. The design reuse may be at the component (model) level or may be a higher abstraction such as an architectural entity. By comparing the components, architectural fragments, and the architectures as a whole, the similarities and differences of the structure, information, and semantics become evident and can be identified. These similarities and differences of architectural entities can ultimately lead to a mapping between architectures and their components.

The scope for developing the framework for comparing software architectures was limited to developing object diagrams to show the syntax (structure) of an architecture. Also, the scope included using the axiomatic approach to show the semantics (behavior) of an architecture's components identified in the object diagram. The axiomatic approach includes the definition of an abstract program P that maps the state space of some architectural entity to another state space.

8.1.2 Specific Conclusions About Software Architectures. The framework identified above was used to analyze the software architectures of VHDL, MetaH, μRapide, IBM ADAGE and OCU. The results of the structural analysis of the architectures revealed:

1. All architectural components have some way to retain their state information. In the lowest-level constructs, the state information is retained in attributes; in higher-level constructs, state information is captured in the subordinate objects, as well as attributes.
2. All components of the different architectures have a way to change state; none are static. Some architectures use operations to change state, some use an update algorithm or an execution path, while others employ a transform.
3. All of the architectures incorporate events and event processing.
4. All constructs have a way to interface with their environment. Some components get their information from inputs, while others get their information from events.
5. All the software architectures analyzed employ a layered architecture. The layered architecture results in different levels of abstractions, with the higher-level abstractions requesting services from lower-level abstractions.

A semantic comparison of the software architectures revealed the following:

1. There does not exist a one-to-one equivalence between any of the architectural entities among the architectures. The transformation process P performs the transformation from state space to state space differently for the components of each architecture.

2. The lowest-level constructs of an architecture only change the state of their local attributes and output values.

3. The lowest-level constructs are passive or reactive in nature.

4. Architectural entities composed of lower-level constructs change state based on the state changes in their subordinate constructs. If an entity has any local state information, this local information changes as a result of a transformation. Each level of abstraction within an architecture is only able to modify the state of components directly subordinate. In effect, the scope of change is limited to one level at a time.

8.2 General Conclusions About Software Architectures

The framework developed as part of this thesis effort had some strong points and some weak points. The strong points are that the architectural entities of a software architecture are readily evident as a result of the object diagram, and the variants and invariants of an entity are evident as a result of the semantics presented using the axiomatic approach. This framework showed that semantically and syntactically there was no direct mapping between architectural entities of different architectures. However, the framework highlights the variant information of an entity. If any type of transformation is to be possible, the information and behavior of an entity identified by this framework must be preserved as a result of a transformation.

Also, this framework highlighted that several common elements seem to be included in all the software architectures. The common elements identified in this research effort may possibly lead to the development of a meta-model for software architectures.

Finally, the framework characterized some aspects (or properties) of architectures that are desirable. While no metrics currently exist for evaluating software architectures, this research is an initial step in identifying architectural properties and corresponding metrics for characterizing software architectures.

A weak point of this research is that it does not give any insight into the development of the abstract program P for the axiomatic approach. It does present the precondition on the state space of an entity and the postcondition that need to be maintained on the variants as a result of the state-space transformation process of P, but it does not provide details of how to derive P.

As a result of developing the framework for comparing the architectures and performing the comparisons of the various software architectures, it can be concluded that the OCU model has all the elements necessary to allow Architect to mature into a domain-oriented application composition and generation system that is able to accept system specifications from an application specialist and produce verifiable software. The syntax is similar to other successful software architectures and the semantics are comparable to all the other architectures as well.

8.2.1 Incorporation of Event Driven Capability into Architect. A need to incorporate event management, handling, and processing into Architect was identified by previous research efforts. Since the initial implementation of Architect employs the OCU architectural model, the inclusion of events and the associated processing requirements had to conform to the OCU model as specified by the SEI. However, SEI documentation on the OCU model makes no mention of events or event processing. Therefore, I had to extrapolate the intended meaning of the OCU model for the inclusion of events. The scope of this work was limited to designing the event management, processing, and handling capabilities of the subsystems and primitives within Architect. The operating environment of the application executive was addressed in a separate effort.

The following was accomplished as a result of incorporating events into Architect:

1. The type of events and their corresponding formats were identified and implemented within the architecture of the OCU model.

2. Event persistence was accomplished via the InEvent and OutEvent areas attached to each subsystem.

3. In keeping with the OCU spirit as intended by the SEI, subsystems consume events, as this falls in the purview of "locus of mission". After consuming the event, a subsystem invokes a primitive to supply the service intended by the event.

4. The flow of events through the subsystem structure was identified. Subsystems needed to be able to interrogate events to determine the event's target. After determining the event's target, a subsystem processes the event or routes it to a subordinate subsystem.

5. In preserving the sequential execution model as implemented by (3, 20), Architect now has two execution functions; one execution function simulates non-event-driven sequential processing and the other simulates an event-driven model.

6. New functions were implemented within Architect to generate SetState, Update, Transmit, and Remove events.

7. The semantic checks were updated to determine the mode of execution and perform additional checks as required.

8. The import and export areas of subordinate subsystems were consolidated to the import and export areas of the highest-level subsystem. With the consolidation of the imports and exports, the information of an import or export object was expanded to include the owning subsystem of the import or export and a routing scheme to locate the primitive that produced or consumed the particular import or export.

With the inclusion of events into Architect, Architect is closer to allowing the simulation of models that require concurrent behaviors. This allows for expansion to new domains that require concurrency to more realistically represent real world entities.

8.3 Recommendations for Future Research

The following is a list of items that were not addressed as part of this research effort and should be addressed in future research efforts:

1. Further validation of event-driven capabilities. The event-driven simulation capability was initially demonstrated using the event-driven circuits domain. With the expansion of Architect's technology base as a result of (27, 26), these domains can be transformed into event-driven domains to further validate the implementation of events within Architect.

2. Incorporation of a "mixed-mode" execution capability in the subsystems. The initial implementation of events into Architect allows only one mode of execution for a developed application. There are circumstances where we want some independent subsystems of an application to execute in a non-event-driven sequential mode while others execute in an event-driven mode.

3. Concurrent simulation. The event-driven simulation capability within Architect allows for only a single thread of control within an application. Expand Architect to include an event-driven concurrent simulation capability with multiple threads of control. A decision will have to be made as to what level of nesting will be allowed for the concurrent processing of subsystems. The level of concurrency must determine whether only the top level independent subsystems execute concurrently, or whether subsystems subordinate to the top level subsystems execute concurrently.

4. Causality dependency. The initial event-driven capability incorporated within the definition of the controller of a subsystem does not check for any causality dependencies. The subsystem controller should be modified to allow for the specification of causal dependencies on the order of events that must occur prior to the execution of the subsystem. These causality dependencies prevent the execution of a subsystem if all the required information or dependencies between independent subsystems are not met.

5. Modeling of real-time systems. Architect does not employ any type of time constraints in the execution of an application. Expand Architect to include the modelling of domains that incorporate the use of real-time constraints.

6. Architectural transformation. This research identified components of alternative architectures along with the associated precondition and postcondition for their state space. Future research should be done on transforming an architectural component and having Architect employ the component within one of its compositions. This transformation should be behavior-preserving such that the Architect meets the entity's preconditions, and after execution, Architect satisfies the entity's postconditions.

7. Meta-Model for architectures. The framework developed as part of this research identified structural components common to all five of the software architectures analyzed. Research needs to be continued on possibly identifying a meta-model for software architectures. The meta-model will be an aid in the transformation of components between architectures.

8. Basis for an architecture metric system. Another result of this research was the identification of some characteristics that are necessary within any software architecture. The two characteristics identified were the encapsulation of state and information, and the limitation of the scope of control of an architectural entity. Further research needs to be done on identifying metrics for evaluating software architectures.

8.4 Concluding Remarks

By identifying a framework for analyzing software architectures, the similarities and differences among architectures were identified. With these qualities evident, a better assessment can be made as to whether an entity or component from another architecture or domain is suitable for inclusion within a system or domain being developed.

Architect brings software engineering a step closer to being a true engineering discipline. It contains a public technology base of certified components, and it provides a means to verify a design (through behavior modeling) prior to full scale development. It provides the capability to model behaviors of objects that are sequential in nature, as well as being able to model the relationships between independent objects. With the framework presented in this thesis, Architect in the near future may be able to incorporate models and components of other domains and architectures directly; thus, Architect may provide for the reuse of designs that cross domain boundaries. All of the aforementioned capabilities are needed to position the software industry to attain the "megaprogramming" environment needed to allow software systems to be developed on time, within budget, error-free and most importantly, meeting user's requirements.

Vita

Warren Evan Gool was born on April 8, 1961 in Goldsboro, North Carolina. He finished his secondary schooling at Goldsboro High School, graduating in 1979. Throughout his schooling, Warren was active in the Boy Scouts; where in April 1978, he achieved scoutings highest award, the distinguished Eagle Scout.

He started his undergraduate education in the fall of 1979 at North Carolina State University. Also while attending NCSU, he was inducted into Phi Eta Sigma Honorary Fraternity. He graduated from NCSU in December 1983.

Lt Gool came on active duty February 15, 1984 when he reported at Tyndall AFB, Florida. He was assigned as a software systems analyst for the 4702nd Computer Services Squadron (CPUSS), a multi-national organization responsible for developing and maintaining the software for the Joint Surveillance System (JSS). While assigned to the 4702nd CPUSS/RSSF, he developed operating system software, Diagnostic system software, and system level test plans and procedures.

In June 1988, Captain Gool accepted a position at NORAD, managing the software configuration for JSS. He organized and scheduled the agenda for the Computer Program Configuration Sub-Board (CPCSB), as well as identified situational assessment information requirements for NORAD's Air Defense Operations Center.

Captain Gool transferred to Wright-Patterson AFB in May 1992 to attend AFIT to complete a Masters of Science (Computer Systems).

Permanent address: 8175 Trafalger Dr
Colorado Springs, CO 80920

Bibliography

1. Agrawala, Ashok, et al. "DSSA for Intelligent Guidance, Navigation and Control." *1992 IEEE Symposium on Computer-Aided Control System Design (CACSD).* 110–116. IEEE Control Systems Society, 1992.

2. Allen, Rober and David Garlan. *A Formal Approach to Software Architectures.* Technical Report, Carnegie Mellon University, February 1992.

3. Anderson, Cynthia Griffin. *Creating and Manipulating Formalized Software Architectures to Support a Domain-Oriented Application Composition System.* MS thesis, AFIT/GCS/ENG/92D, School of Engineering, Air Force Institute of Technology(AU), Wright-Patterson AFB, OH, December 1992.

4. Bailor, Major Paul D. "The Big Picture, Software Engineering Becoming More Like Traditional Engineering," *CSCE793 - Formal-Based Methods in Software Engineering* (February 1993).

5. Bailor, Paul D. and others. "Formalization and Visualization of Domain-Specific Software Architectures," *AAAI-92 Workshop on Automated Software Design, AAAI Conference*, 6–11 (1992).

6. Batory, Don and Sean O'Malley. "The Design and Implementation of Hierarchical Software Systems with Reusable Components," (January 1993).

7. Coad, Peter and Edward Yourdon. *Object-Oriented Analysis (Second Edition).* Englewood Cliffs, New Jersey: Pentice Hall, Inc, 1991.

8. Coglianese, Lou, et al. "DSSA: Navigation, Guidance, and Flight Driector Design and Development." *1992 IEEE Symposium on Computer-Aided Control System Design (CACSD).* 102–109. IEEE Control Systems Society, 1992.

9. Cossentine, Jay. *Developing a Sophisticated User Interface to Support Domain-Oriented Application Composition and Generation Systems.* MS thesis, AFIT/GCS/ENG/93D, School of Engineering, Air Force Institute of Technology(AU), Wright-Patterson AFB, OH, December 1993.

10. D'Ippolito, Richard S. "Using Models in Software Engineering." *Proceedings: TRI-Ada '89.* 256–265. New York, NY: Association of Computing Machinery, Inc., 1989.

11. Dromey, Geoff. *Program Derivation, The Development of Programs From Specifications.* Reading, Massachusetts: Addison-Wesley Publishing Company, 1991.

12. Fenton, N E. *Software Metrics: A Rigorous Approach.* Chapman and Hall (Van Nostrand Reinhold), 1991.

13. Fischer, Charles N and Rchard J LeBlanc. *Crafting a Compiler.* Menlo Park, California: thebenjamin/Cummings Publishing Company, Inc, 1988.

14. Lee, Kenneth J. and others. *Model-Based Software Development (Draft).* Technical Report CMU/SEI-92-SR-00, Software Engineering Institute, December 1991.

15. Lipsett, Roger, et al. *VHDL: Hardware Description and Design.* Norwell, Massachusetts: Kluwer Academic Publishers, 1989.

16. Lowry, Michael R. "Software Engineering in the Twenty-first Century." *Automating Software Design*, edited by Michael R. Lowry and Robert D. McCartney. 627–654. Menlo Park, CA: AAAI Press/MIT Press, 1991.

17. Luckham, David C. and James Vera. "μRapide: An Executable Architecture Definition Language," (April 7 1993).

18. Mettala, Lieutenant Colonel Erik and Marc H. Graham. "The DSSA Program," *CrossTalk - The Journal of Defense Software Engineering*, *37*:19–21 (October 1992).

19. Prieto-Diaz, Ruben. "Domain Analysis for Reusability," *IEEE Computer*, 63–69 (March 1987).

20. Randour, Captain Mary Anne. *Creating and Manipulating a Domain Specific Formal Object Base*. MS thesis, AFIT/GCS/ENG/92D, School of Engineering, Air Force Institute of Technology(AU), Wright-Patterson AFB, OH, December 1992.

21. Rich, Elain and Devin Knight. *Artificial Intelligence, Second Edition*. New York: McGraw-Hill, Inc, 1991.

22. Rumbaugh, James and others. *Object-Oriented Modeling and Design*. Englewood Cliffs, New Jersey: Pentice Hall, Inc, 1991.

23. Silberschatz, Abraham and others. *Operating System Concepts, Third Edition*. Reading, Massachusetts: Addison-Wesley Publishing Company, 1991.

24. Vestal, Steve. *A Cursory Overview and Comparison of Four Architecture Description Languages*. Technical Report, Honeywell Systems and Research Center MN65-2100, February 1993.

25. Vestal, Steve. *Software Programmer's Manual for the Honeywell Aerospace Compiled Kernal (MetaH Language Reference Manual) (Draft)*. Technical Report a.5, Honeywell Systems and Research Center MN65-2100, June 1993.

26. Waggoner, Robert. *Domain Modeling of Time-Dependent Systems*. MS thesis, AFIT/GCS/ENG/93D, School of Engineering, Air Force Institute of Technology(AU), Wright-Patterson AFB, OH, December 1993.

27. Warner, Russel. *A Method for Populating the Knowledge Base of AFIT's Domain-Oriented Application Composition System*. MS thesis, AFIT/GCS/ENG/93D, School of Engineering, Air Force Institute of Technology(AU), Wright-Patterson AFB, OH, December 1993.

28. Weide, Lieutenant Timothy. *Developement of a Visual System Interface to Support a Domain-Oriented Application Composition System*. MS thesis, AFIT/GCS/ENG/93M, School of Engineering, Air Force Institute of Technology(AU), Wright-Patterson AFB, OH, March 1993.

29. Welgan, Robert L. *Domain Analysis and Modeling of a Model-Based Software Executive*. MS thesis, AFIT/GCS/ENG/93D, School of Engineering, Air Force Institute of Technology(AU), Wright-Patterson AFB, OH, December 1993.

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE December 1993	3. REPORT TYPE AND DATES COVERED Master's Thesis

4. TITLE AND SUBTITLE ALTERNATIVE ARCHITECTURES FOR DOMAIN-ORIENTED APPLICATION COMPOSITION AND GENERATION SYSTEMS	5. FUNDING NUMBERS
6. AUTHOR(S) Warren E. Gool	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Air Force Institute of Technology, WPAFB OH 45433-6583	8. PERFORMING ORGANIZATION REPORT NUMBER AFIT/GCS/ENG/93D-11
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Capt Rick Painter 2241 Avionics Circle, Suite 16 WL/AAWA-1 BLD 620 Wright-Patterson AFB, OH 45433-7765	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
11. SUPPLEMENTARY NOTES	
12a. DISTRIBUTION/AVAILABILITY STATEMENT Distribution Unlimited	12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This thesis presents a formalized framework for comparing the structure and semantics of software architectures. The framework uses object diagrams for analyzing the structure of the architectures and the axiomatic approach for analyzing the semantics. This framework is used to compare the Object Connection Update (OCU) model (developed by the Software Engineering Institute) against four other software architectures: VHDL defined by Lipsett, MetaH defined by Honeywell, μRapide defined by Luckham, and hierarchical software systems as defined by Batory. The goal of the comparison was to evaluate the OCU model for suitability within prototype application composition and generation systems. This research concluded that the OCU model has all the elements necessary for use in application composition and generation systems. Additionally, the framework identified several common elements in all the software architectures. These common elements may lead to the development of a "meta-model" for software architectures.

14. SUBJECT TERMS Software Engineering, Automatic Programming, Systems Engineering, Computer Program Verification, Knowledge Based Systems, Computer Architecture	15. NUMBER OF PAGES xxx
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL